Patent Office
Canberra

I, KIM MARSHALL, MANAGER PATENT OPERATIONS hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. PP 7024 for a patent by SILVERBROOK RESEARCH PTY LTD filed on 09 November 1998.

WITNESS my hand this
Twenty-third day of September 1999

KIM MARSHALL
MANAGER PATENT OPERATIONS

AUSTRALIA

Patents Act 1990

PROVISIONAL SPECIFICATION

Applicant(s):

SILVERBROOK RESEARCH PTY LTD

Invention Title: Image Creation Method and Apparatus (ART77)

The invention is described in the following statement:

IMAGE CREATION METHOD AND APPARATUS (Art 77)

The present invention relates to printing systems and in particular discloses a print processor (ICP) for processing images to be printed out.

Summary of the Invention

It is an object of the present invention to provide for an effective print processor for interconnecting with a computer system.

In accordance with a first aspect of the present invention, there is provided a print processor for interconnecting with a computer system and for the production of output pixel data for driving a printhead to produce an output image, comprising: an interface unit for interconnection with the computer system and the receipt of commands therefrom; a bi-level expansion unit interconnected to the interface unit and adapted to receive compressed data for a single color and to expand the compressed data to provide a pixel level representation of the single color data; a pixel expansion unit interconnect to the interface unit and adapted to receive compressed data for the remaining colors of the output image and to decompress the compressed data of the remaining colors to provide a pixel level representation of the remaining color data; compositing means interconnected with the pixel expansion unit and the bi-level expansion unit and adapted to composite the single color data and the remaining color data together to produce image pixel data.

The print processor can further comprise a halftoning means interconnected with the compositing means and adapted to produce color bi-level data from the remaining color data for compositing by the compositing means.

The bi-level data can be produced utilizing a dither volume and the dither volume can comprise a series of columns of bit data and the columns can be divided into a

predetermined number of segments with each segment comprising the same value bit data.

The printhead can comprise a pagewidth printhead of an array of ink ejection nozzles. The array of ink ejection nozzles eject ink as a result of activation of a thermal bend actuator.

The interface unit can comprise a Universal Serial Bus Interface.

The single color compress data utilizes an edge delta and runlength compression format and the single color compress data can comprise a black color channel.

Further, the processor can be adapted to download one page from the computer whilst printing another.

Brief Description of the Drawings

Notwithstanding any other forms which may fall within the scope of the present invention, preferred forms of the invention will now be described, by way of example only, with reference to the accompanying drawings in which:

Fig. 1 illustrates a first front view of the printer system;

Fig. 2 is perspective view of an open printer system;

Fig. 3 is a rear perspective view of the printer unit;

Fig 4 is a rear perspective view with the ink cartridge removed from the printer unit;

Fig. 5 is a front perspective view with the front cover removed;

Fig. 6 is a perspective, partly in section, illustrating the internal operational portions of the printer unit;

Fig. 7 is a close up exploded perspective of portions of the printer unit;

Fig. 8 is an exploded perspective view, taken from the back, of portions of the printer unit;

Fig. 9 is a view of portions of the printer unit illustrating the roller system;

Fig. 10 illustrates the electronics of the printer unit;

Fig. 11 is a perspective of a print cartridge;

Fig. 12 is a perspective of a small print cartridge with the lid removed;

Fig. 13 illustrates the ink flow within a print cartridge.

Description of Preferred and Other Embodiments

Turning initially to Fig. 1, there is illustrated an initial view of the A4 pagewidth printer of the preferred embodiment when in a closed position. The printer 1 includes a front lid portion 2 which, as illustrated in Fig. 2 can be opened so as to provide for an extended supporting structure. Paper is inserted in the slot 4 so as to rest on the lid 2 and flap 3. After printing has occurred, the output is output along output tray 5. A power button 7 is provided in addition to form feed button 8 and ink status buttons 9.

In Fig. 3, there is illustrated a rear view of the printer 1 which includes a detachable cover 12 so as to allow access to an ink supply cartridge. Also a power input 13 and USB connection 14 is also provided.

As illustrated in Fig. 4, the cover 12 is detachable and includes, detachably mounted thereon, an ink supply cartridge 15 which together are slotted in an aperture in the back of the printer 1. The cartridge 15 is designed to mate with a cartridge interconnect port 18 and a series of ink supply channels 19 upon insertion of the cover 12 into position.

Fig. 5 illustrates the printer 1 with the removal of the front cover so as to reveal internal portions of the printer. The internal portions includes cover portions 20, 21.

In Fig. 6, there is shown the internal printer mechanisms with the outer shell removed in addition to the removal of the portions 20, 21 of Fig. 6. Fig. 6 illustrates a sectional view in that the printhead portions 30 which are shown to include a separate pagewidth micromechanical ink jet printhead 31. A first backing portion 32 providing four colour ink to the printhead constructed with a first level of fine tolerance so as to match those utilized in the printhead 31. A second supporting number 33 is in turn provided having a lower tolerances manufacture. An outer tab casing 34 is also provided. The construction of the printhead cartridge 30 is very similar to that disclosed in the aforementioned Australian Provisional Patent Application No. PP6534 filed 25 October 1998 entitled "Micromechanical Device and Method (IJ46a) which is specifically incorporated herewith by cross-reference.

In Fig. 7 there is illustrated a close up of the left hand side of the arrangement of Fig. 6 with Fig. 8 showing a back left hand side view of the arrangement of Fig. 6. The arrangement includes a first roller 40 which is designed to pinch a page of paper for feeding through the printer system. The paper is fed to second roller 41 which mates with a series of platten surface rollers. A third roller 42 is provided for pinching the output paper so that it is fed across the printhead 31 in the stable manner. A series of gears 45 are provided for appropriate driving of the rollers 40 - 42 under the control of motor 50 (See Fig. 8).

As shown in Fig. 9, which illustrates the operation of the rollers 40, 42 and gearing mechanisms in addition to the stepper motor 50, the roller 41 of Fig. 6 mates with a series of rollers eg. 53 which are flexibly mounted on a platten 52 which also includes an ink jet printhead capping unit 54. The operation of the capping

unit 54 can be described in PCT Patent Application No. AU98/00544, the contents of which are incorporated by cross reference herein. The rollers eg. 53 act to transfer the paper over the platten 52 and capping mechanism 54.

Turning now to Fig. 10, there is illustrated a number of internal electrical control portions of the printer system. The core of the printer system is a ASIC 60 which interacts with a standard memory chip 61 which can comprise a 64 Mbit DRAM. The ASIC chip 60 hereinafter called the ICP is responsible for all processing operations for the printer 1 and interacts with an external system by a USB port 62. A series of capacitors, resistors and power transistors as required are also provided on the circuit board 64 which interconnects with the printhead via the connector cable 65. The front panel switches 66 are also interconnected via cables 67. A paper sensor 68 can also be provided for sensing the presence of paper. The interconnection of the cable 65 to the printhead can be as described in the aforementioned Australian Provisional Patent Specification.

Returning now to Fig. 4, one aspect of the preferred embodiment is that different sized ink cartridges 15 can be inserted in the back of the printer 1. In Fig. 11 there is illustrated an alternative form 70 which is a shortened ink cartridge in comparison with that illustrated in Fig. 4. The ink cartridge 70 includes four colour outputs 71 which supply ink to the printer. Each ink chamber includes an air inlet means eg. 72 which comprises a winding "pipe" which is sealed by means of a hydrophobic tape 73 so as to provide for a single air inlet eg. 74.

Turning now to Fig. 12, there is illustrated the internal portions of the ink cartridge 70 which provides all four ink reservoirs 76 - 79 with each reservoir connected to our corresponding ink output.

Turning now to Fig. 13 there is illustrated ink path from the ink supply cartridge 70 to the printhead cartridge 30. The ink flows out of the ink cartridge 70 along channels formed within member 81 to the printhead 30.

Each ink cartridge eg. 70 of Fig. 13 or 15 of Fig. 4 can further include an authentication chip so as to ensure only valid ink cartridges are utilized in the overall printer. The authentication chip can be similar to that described in the aforementioned PCT patent specification previously incorporated by cross reference. The authentication chip is interrogated by the ICP ASIC 60 to determine whether a valid cartridge is inserted.

A full description of the operation of the ICP ASIC is given in the attached Appendix A.

It would be appreciated by a person skilled in the art that numerous variations and/or modifications may be made to the present invention as shown in the specific embodiments without departing from the spirit or scope of the invention as broadly described. The present embodiments are, therefore, to be considered in all respects to be illustrative and not restrictive.

We Claim:

1. A print processor for interconnecting with a computer system and for the production of output pixel data for driving a printhead to produce an output image, comprising:

an interface unit for interconnection with said computer system and the receipt of commands therefrom;

a bi-level expansion unit interconnected to said interface unit and adapted to receive compressed data for a single color and to expand said compressed data to provide a pixel level representation of said single color data;

a pixel expansion unit interconnect to said interface unit and adapted to receive compressed data for the remaining colors of said output image and to decompress said compressed data of said remaining colors to provide a pixel level representation of said remaining color data;

compositing means interconnected with said pixel expansion unit and said bi-level expansion unit and adapted to composite said single color data and said remaining color data together to produce image pixel data.

2. A print processor as claimed in claim 1 further comprising:

halftoning means interconnected with said compositing means and adapted to produce color bi-level data from said remaining color data for compositing by said compositing means.

3. A print processor as claimed in claim 2 wherein said bi-level data is produced utilizing a dither volume.

4. A print processor as claimed in claim 3 wherein said dither volume comprises a series of columns of bit data and said columns can be divided into a predetermined number of segments with each segment comprising the same value bit data.

5. A print processor as claimed in any previous claim wherein said printhead comprises a pagewidth printhead of an array of ink ejection nozzles.

6. A print processor as claimed in claim 5 wherein said array of ink ejection nozzles eject ink as a result of activation of a thermal bend actuator.

7. A print processor as claimed in any previous claim wherein said interface unit comprises a Universal Serial Bus Interface.

8. A print processor as claimed in any previous claim wherein said single color compress data utilizes an edge delta and runlength compression format.

9. A print processor as claimed in any previous claim wherein said single color compress data comprises a black color channel.

10. A print processor as claimed in any previous claim wherein said processor is adapted to download one page from the computer whilst printing another.

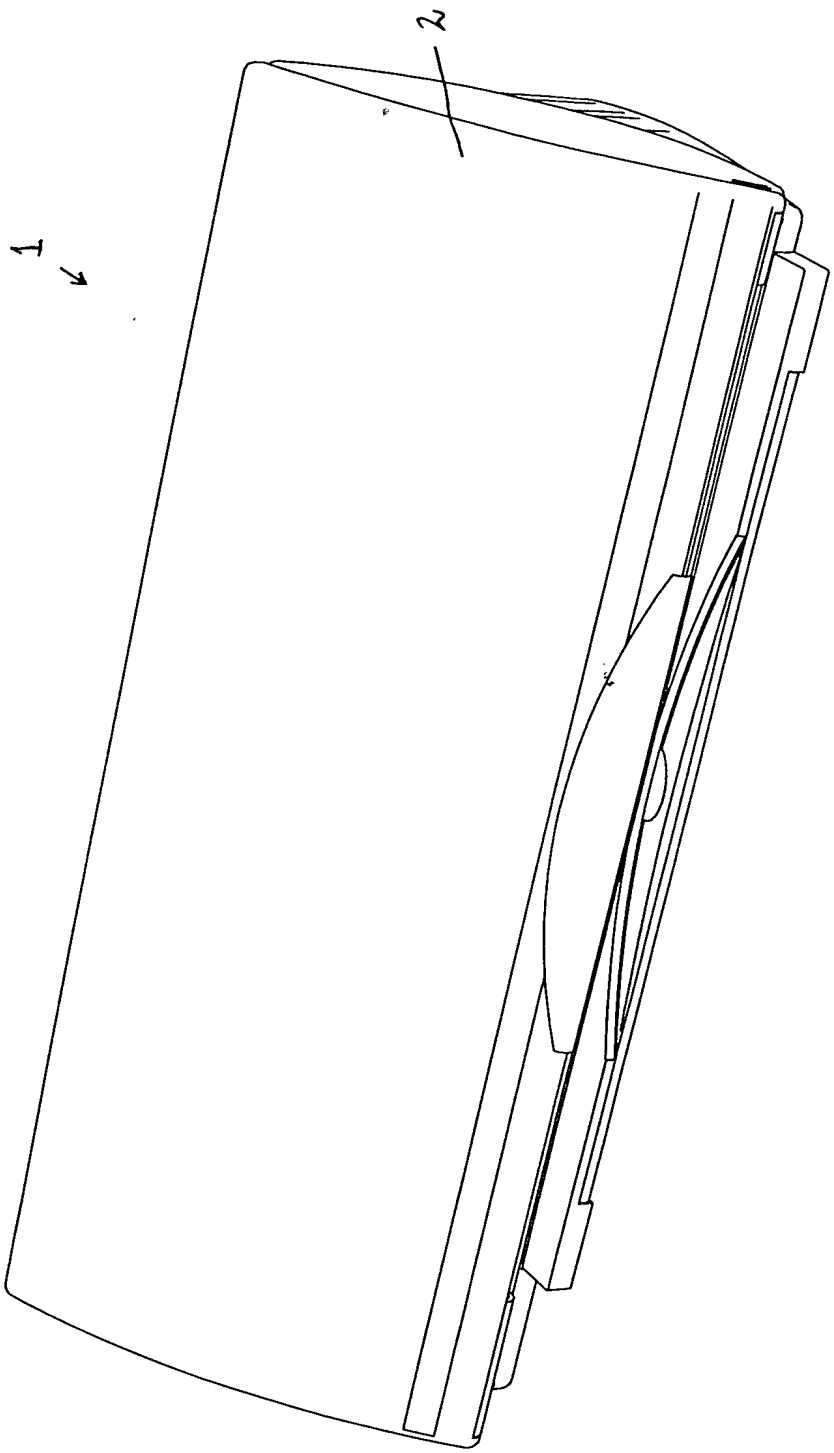
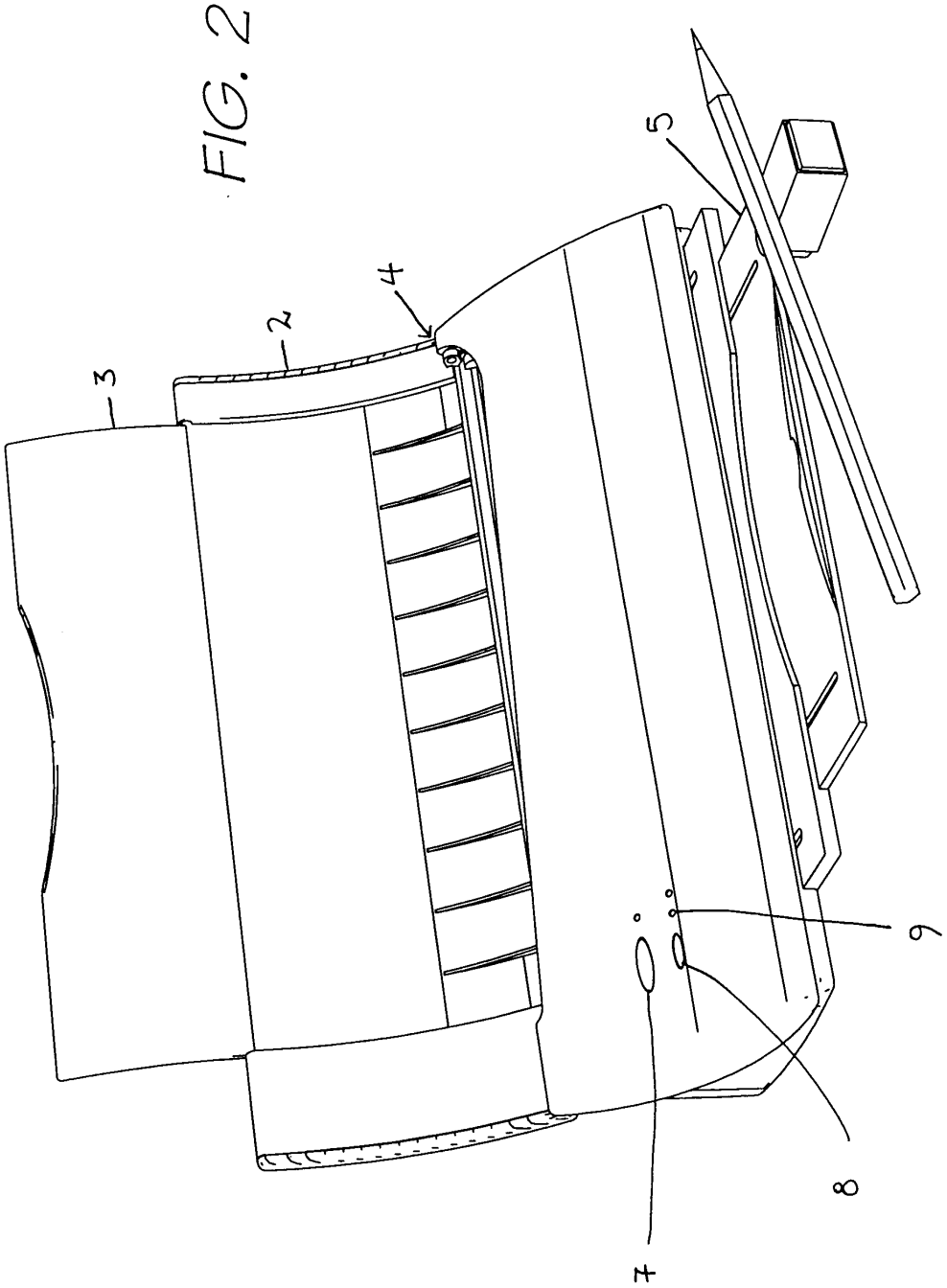


FIG. 1

FIG. 2



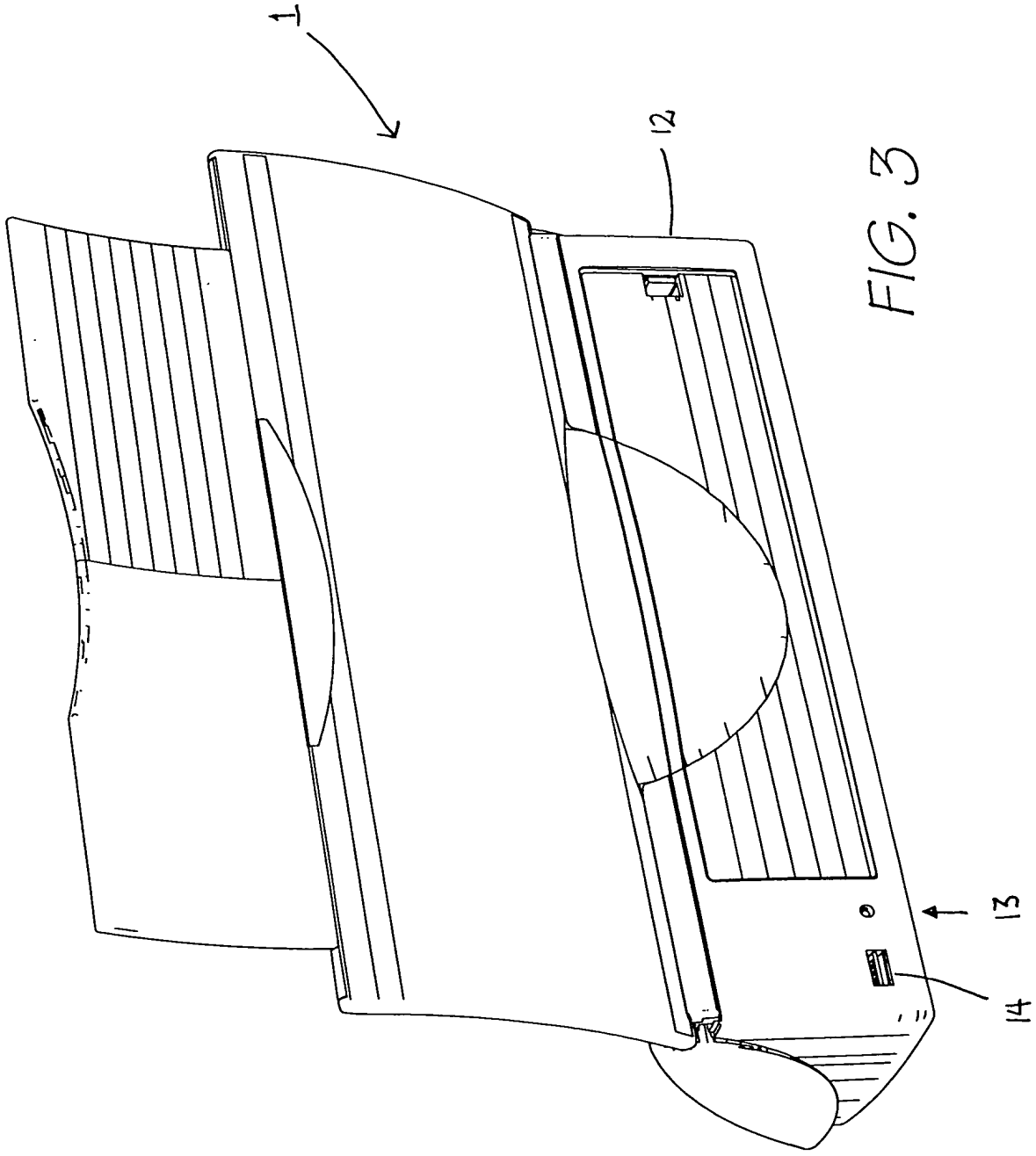


FIG. 3

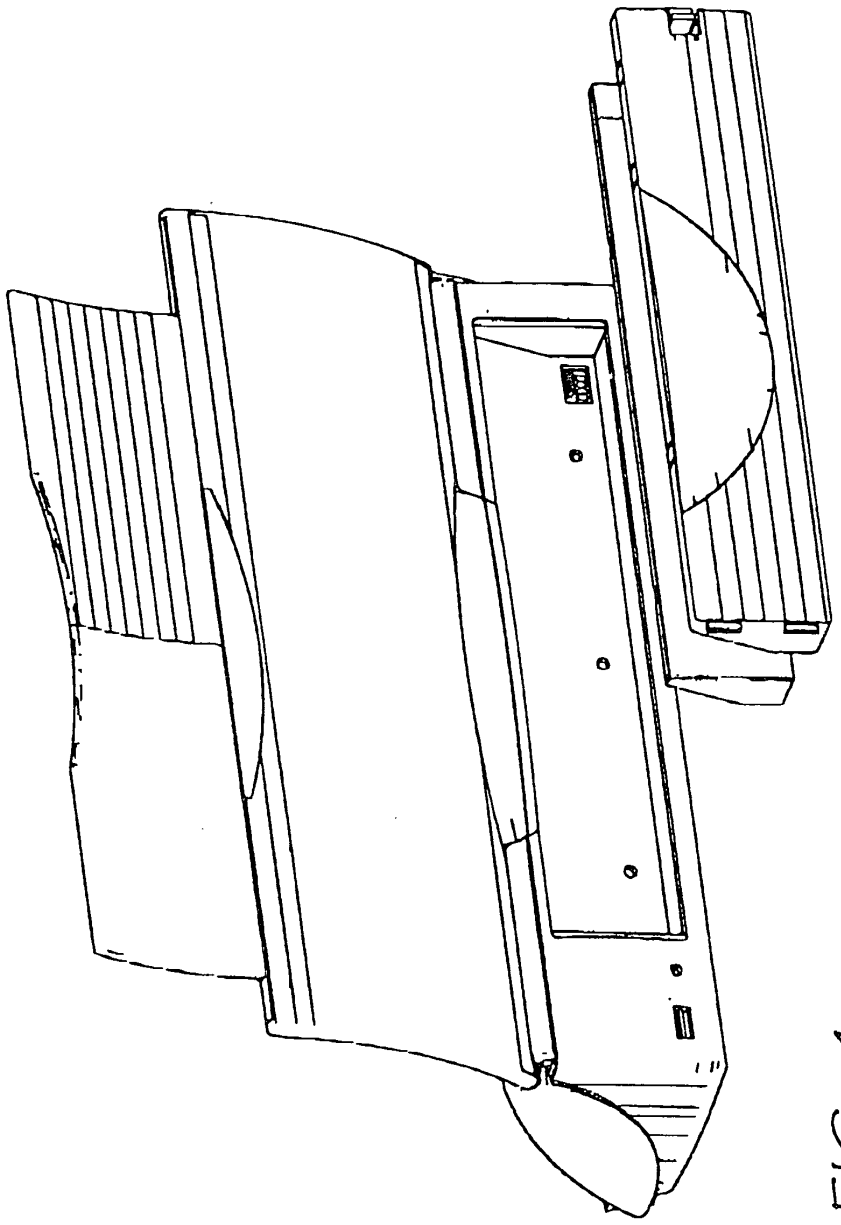


FIG. 4

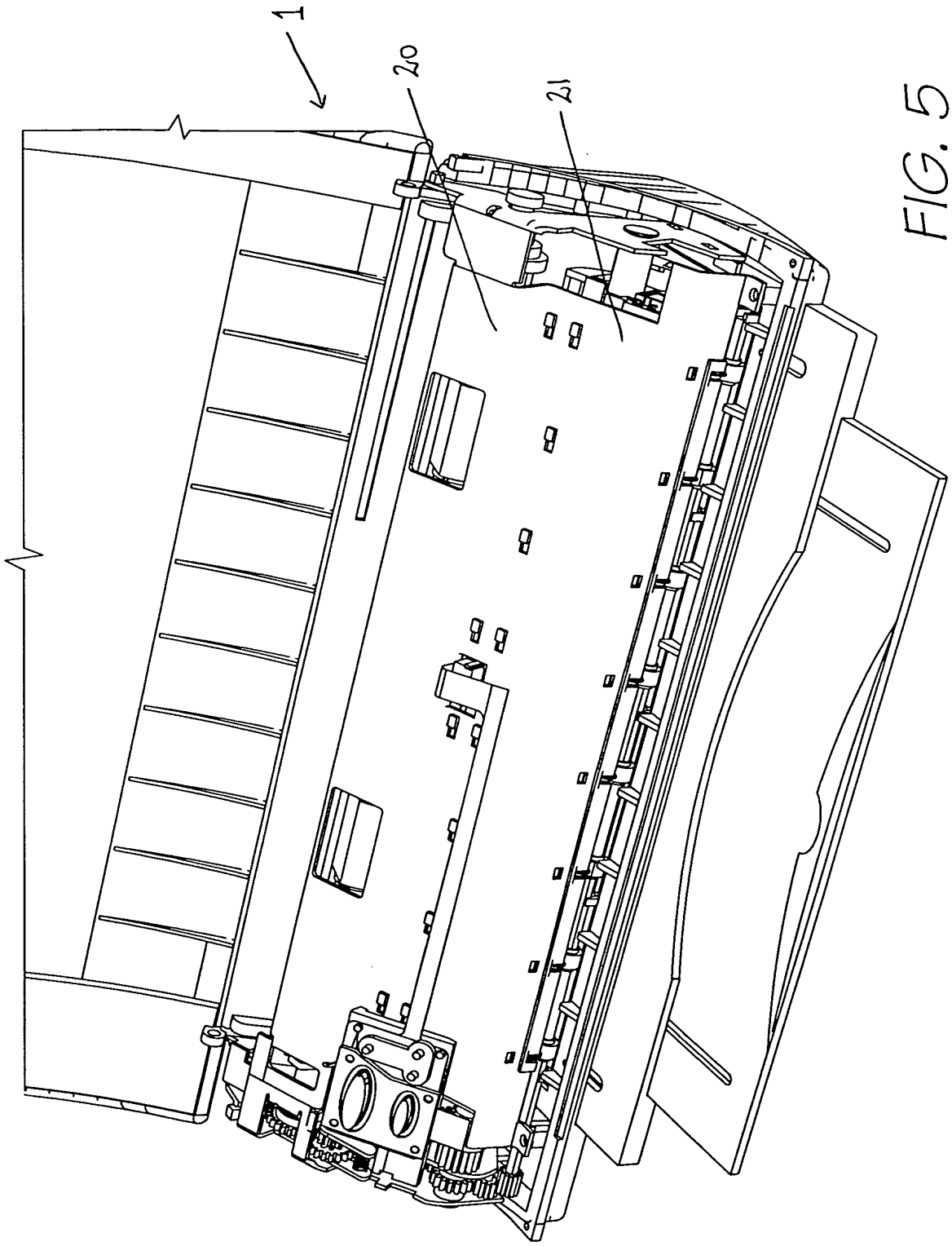


FIG. 5

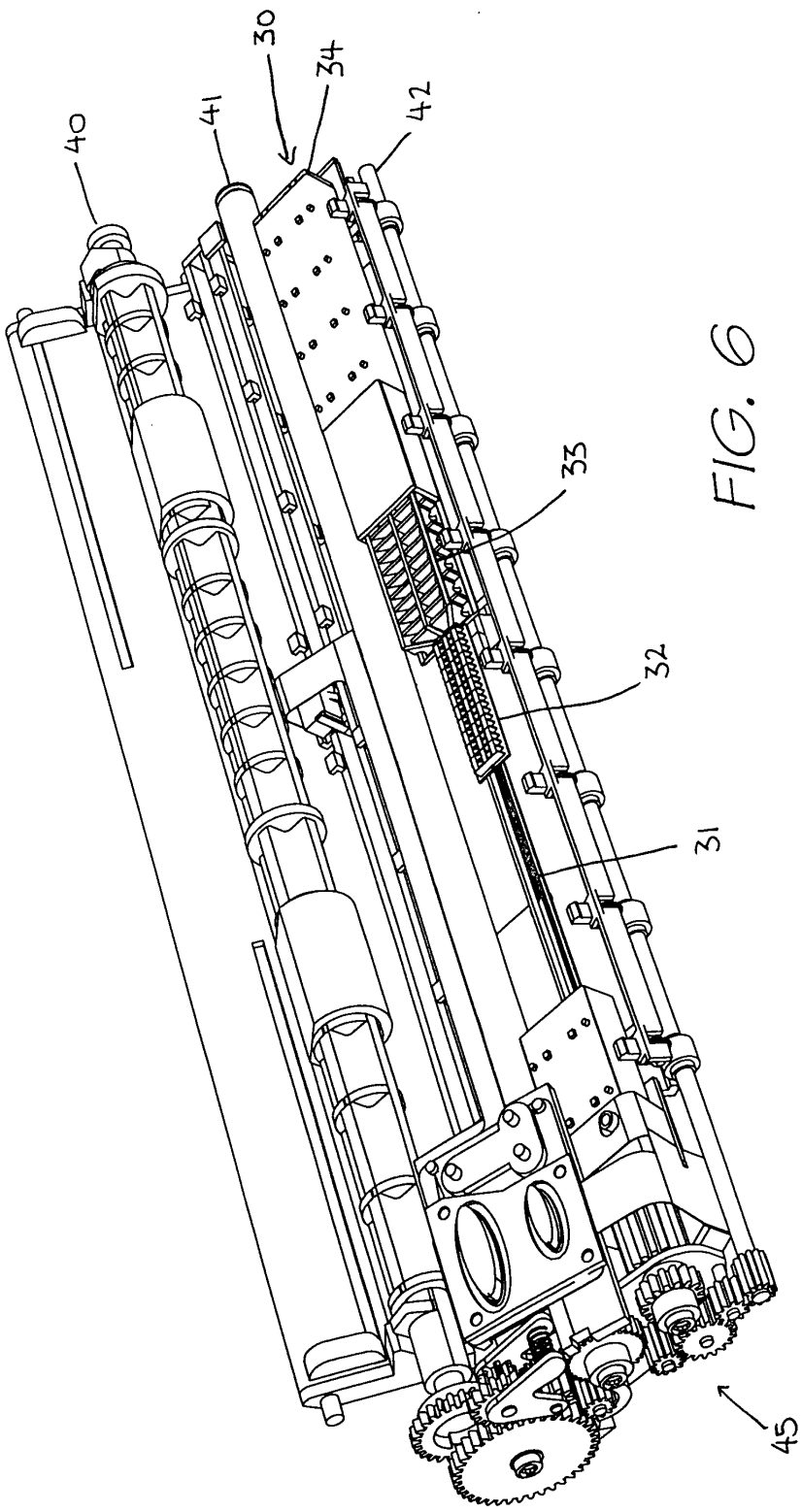


FIG. 6

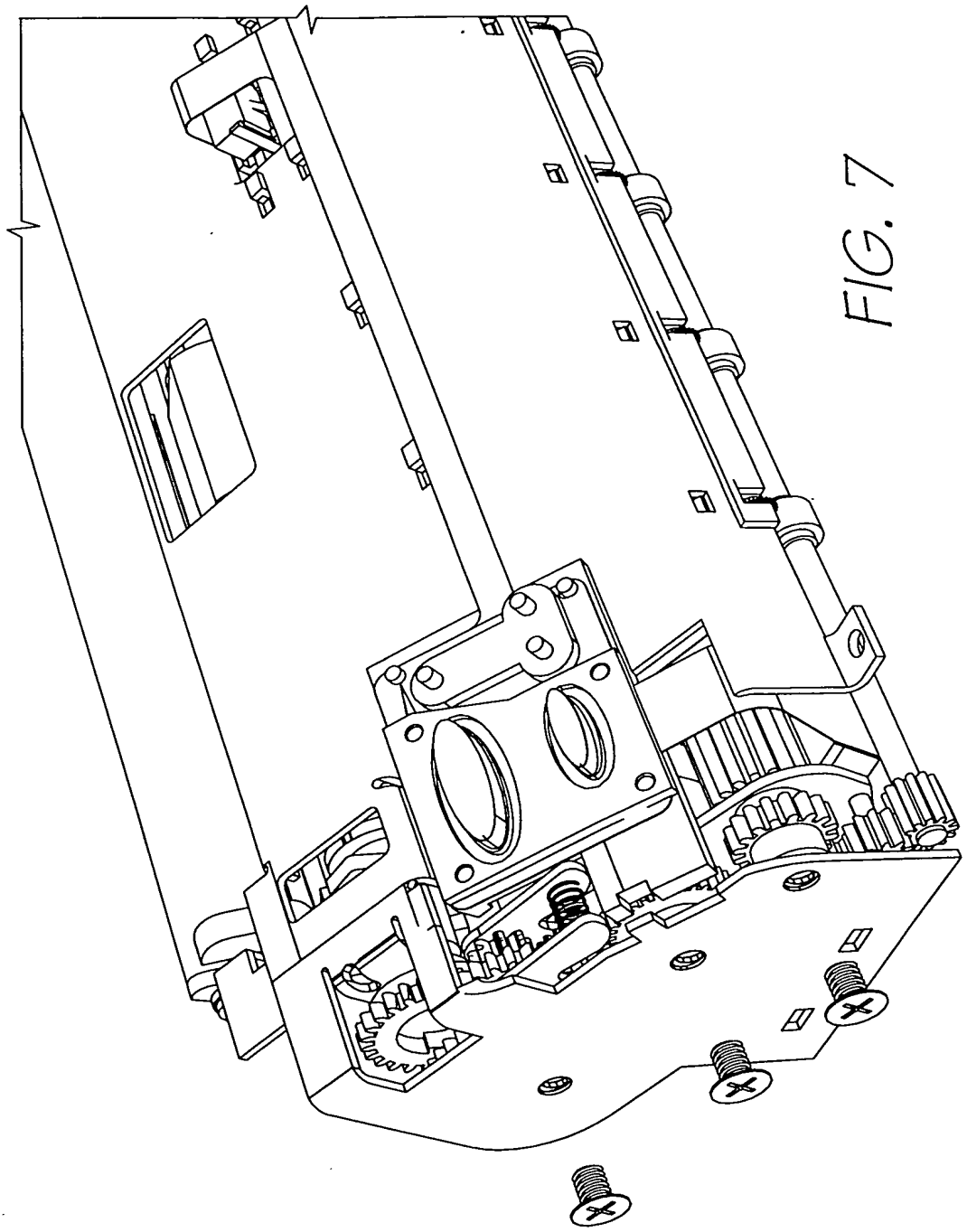


FIG. 7

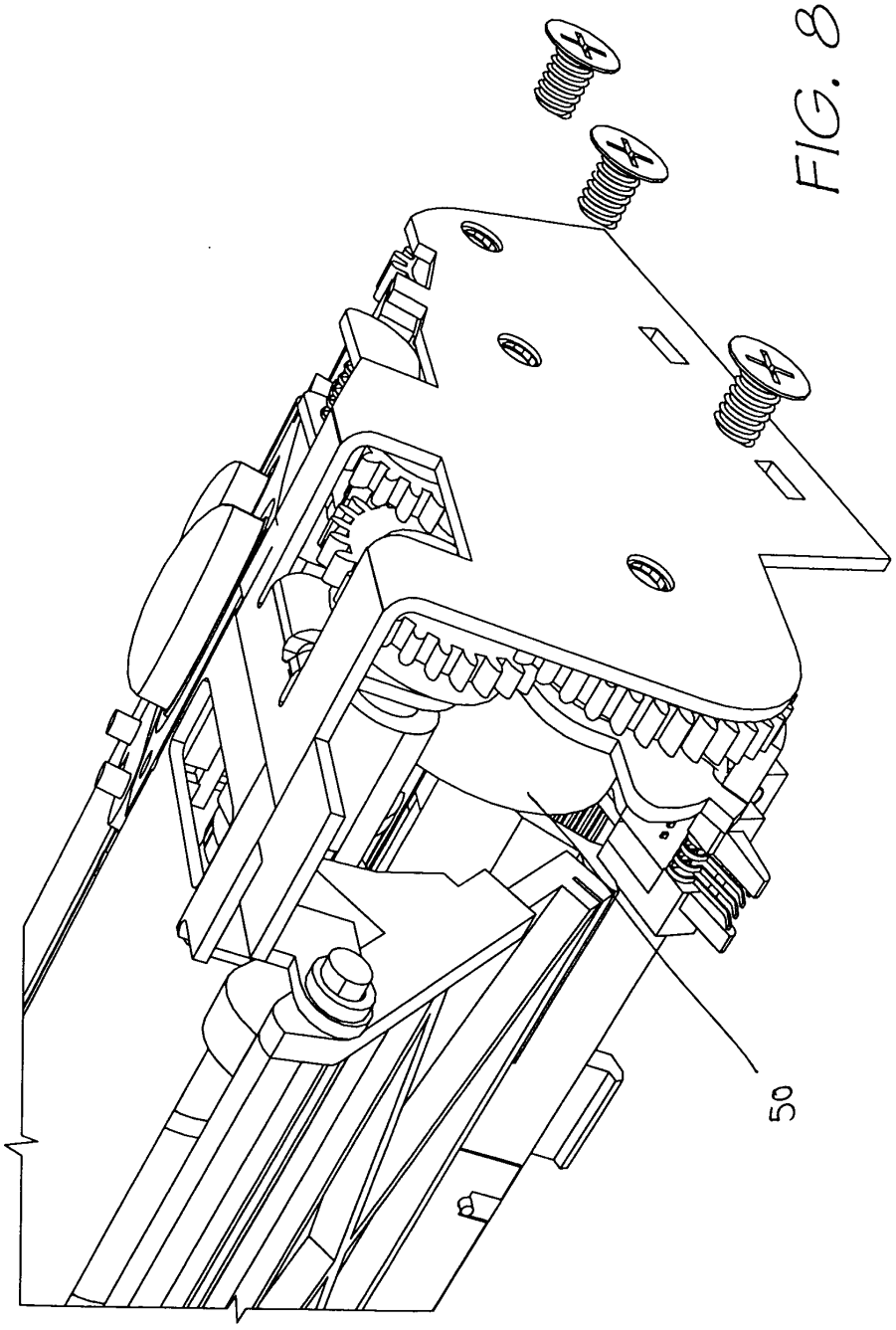


FIG. 8

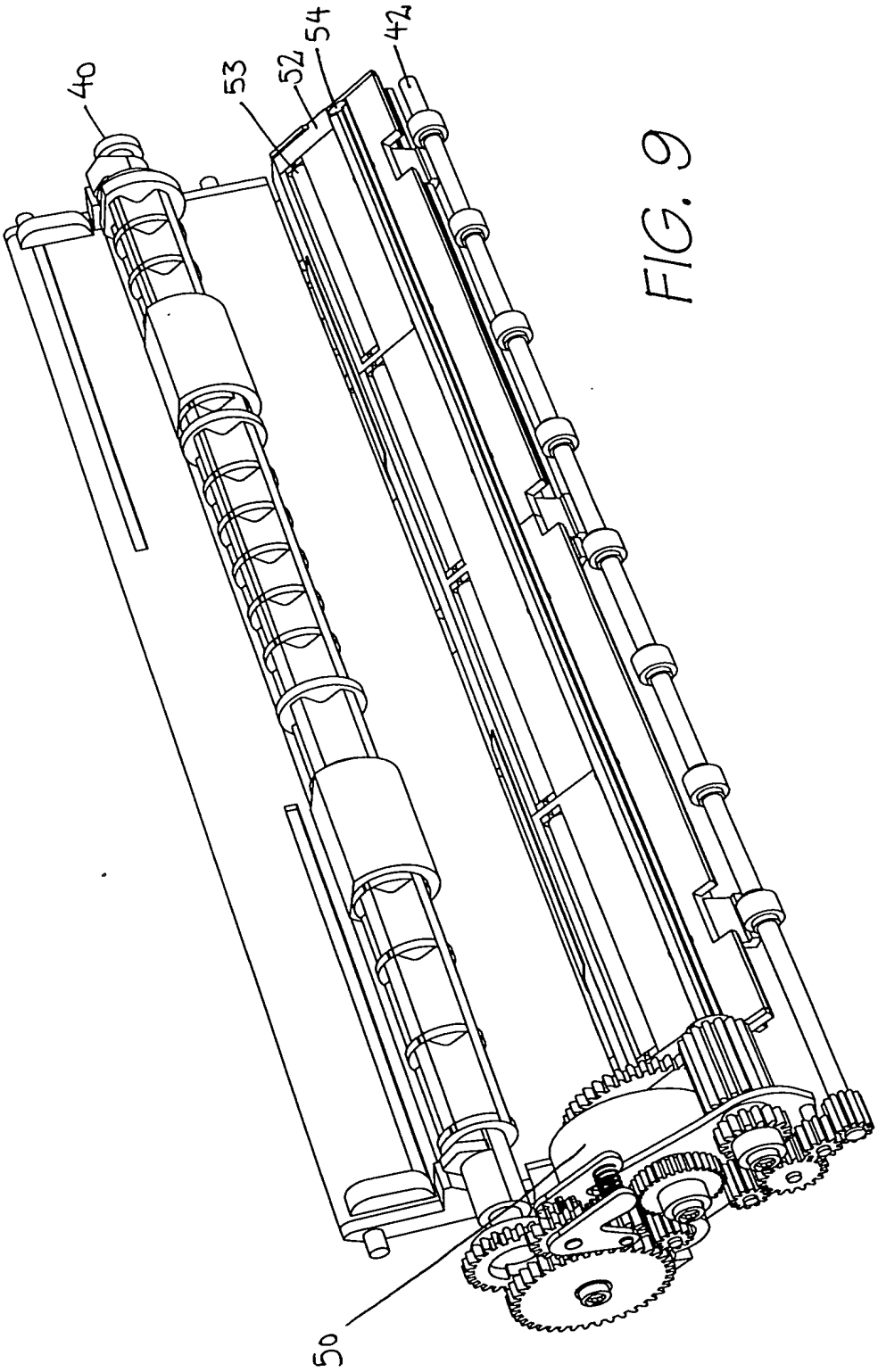


FIG. 9

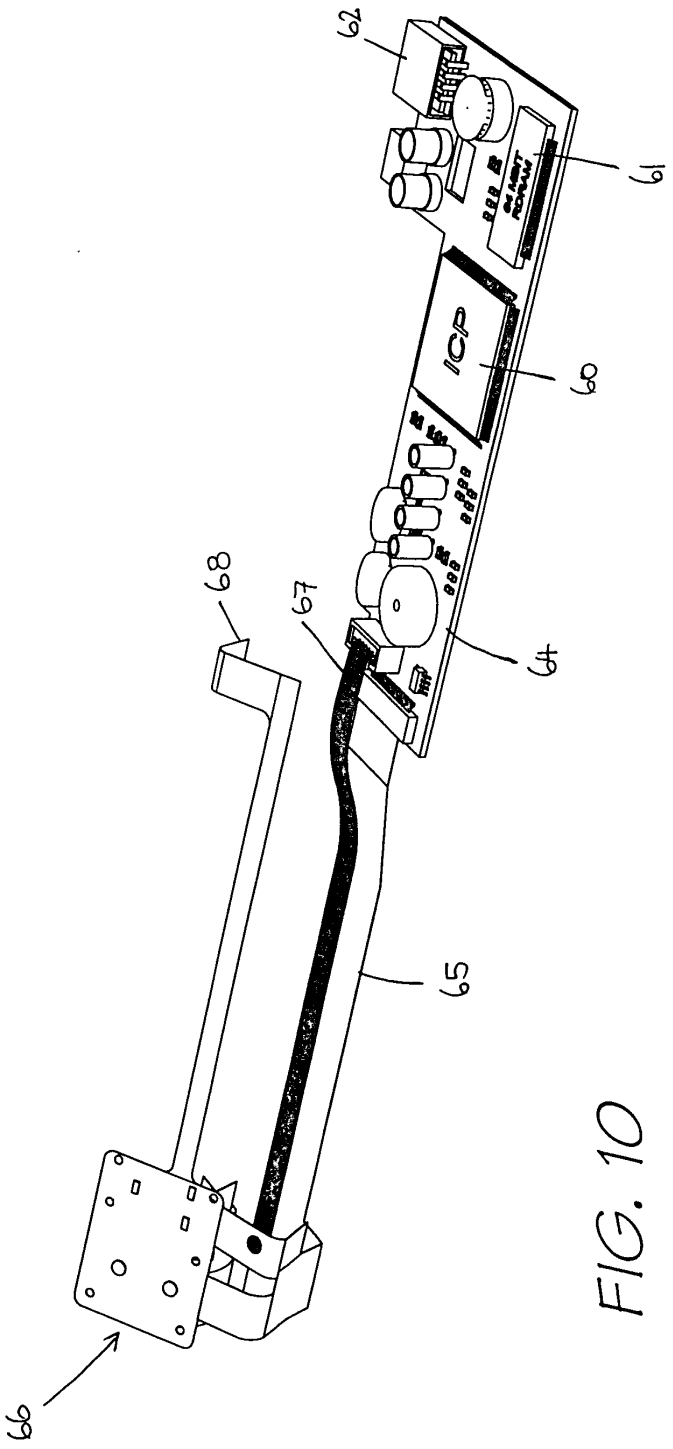


FIG. 10

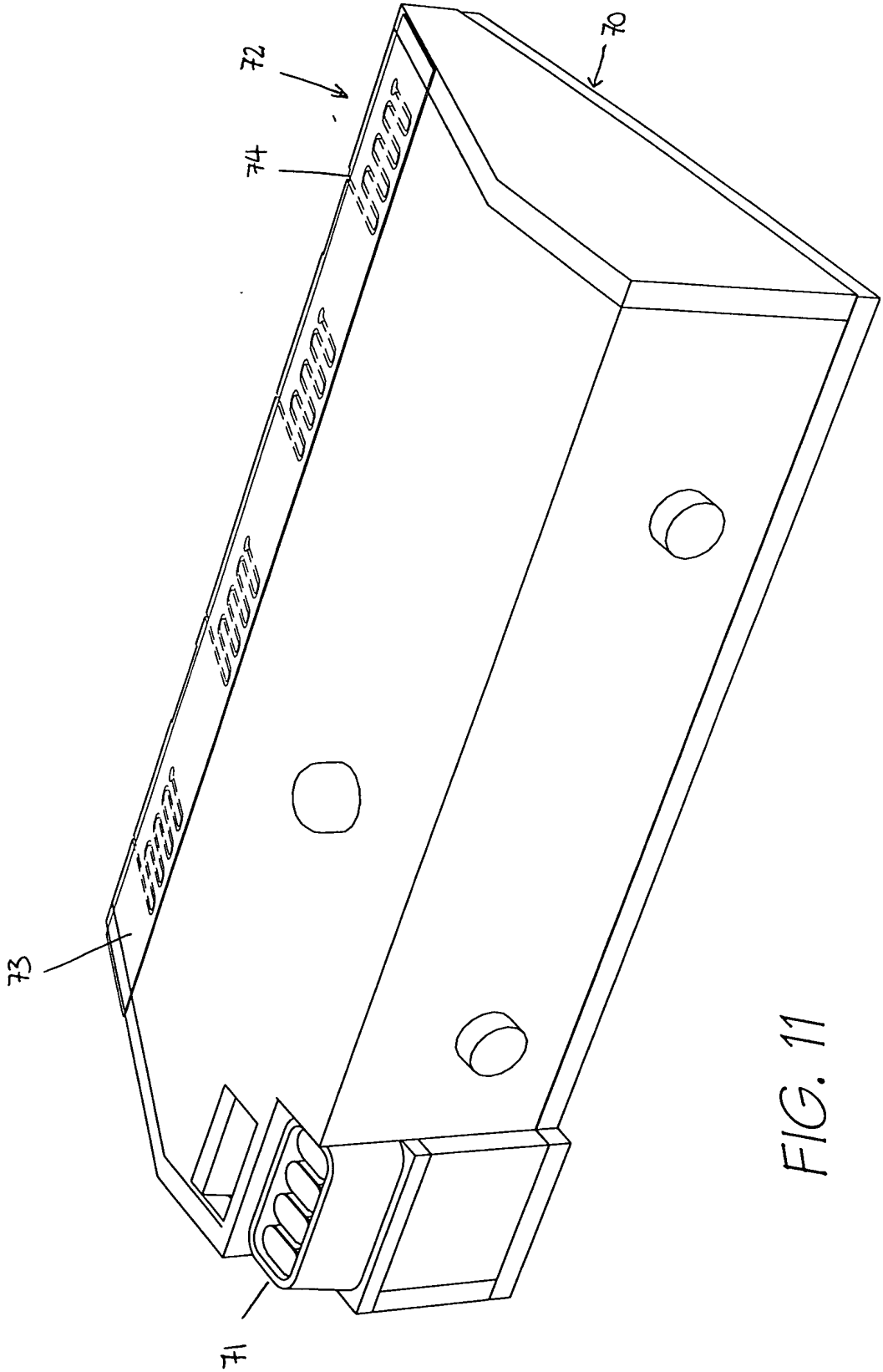


FIG. 11

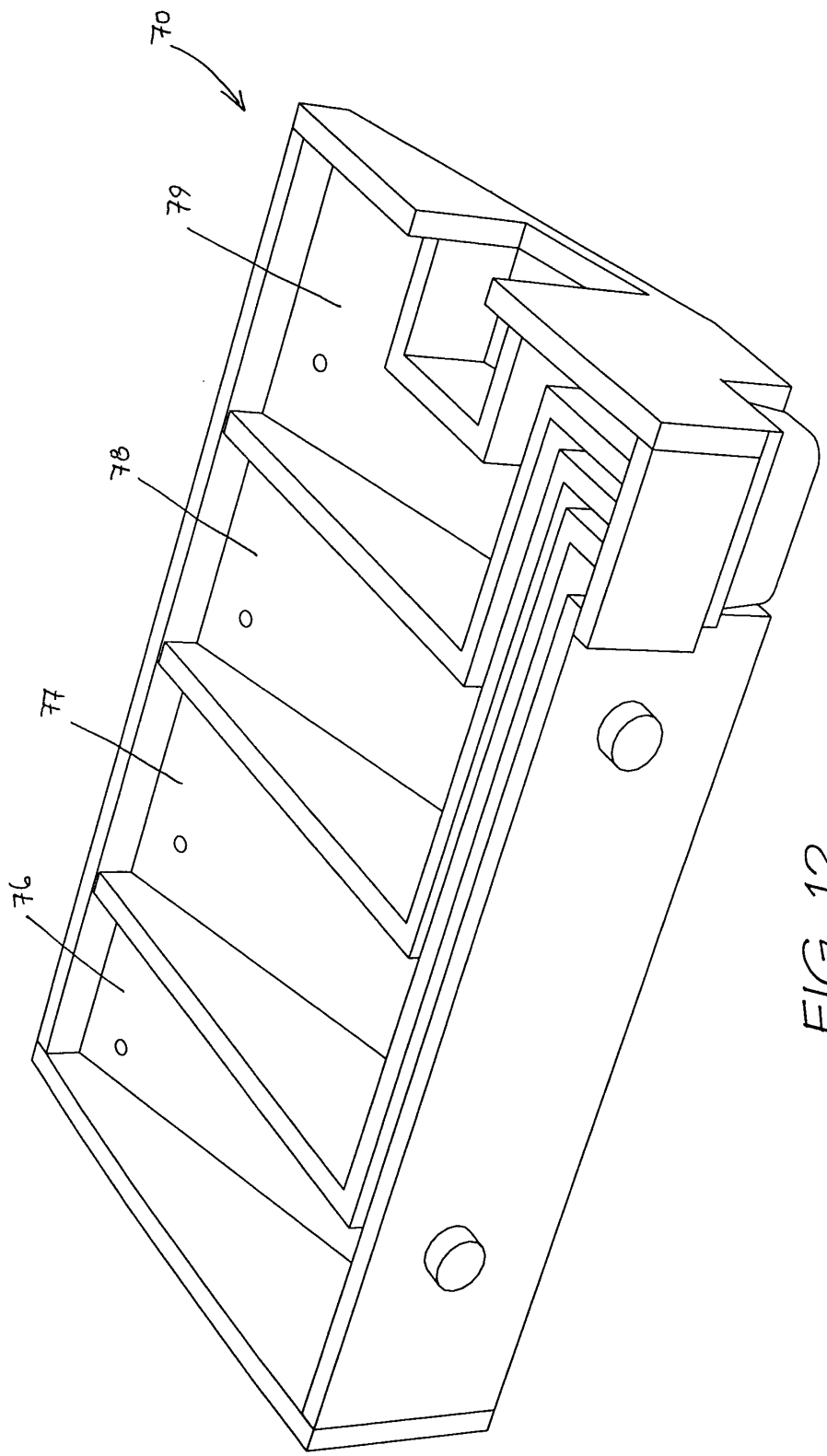


FIG. 12

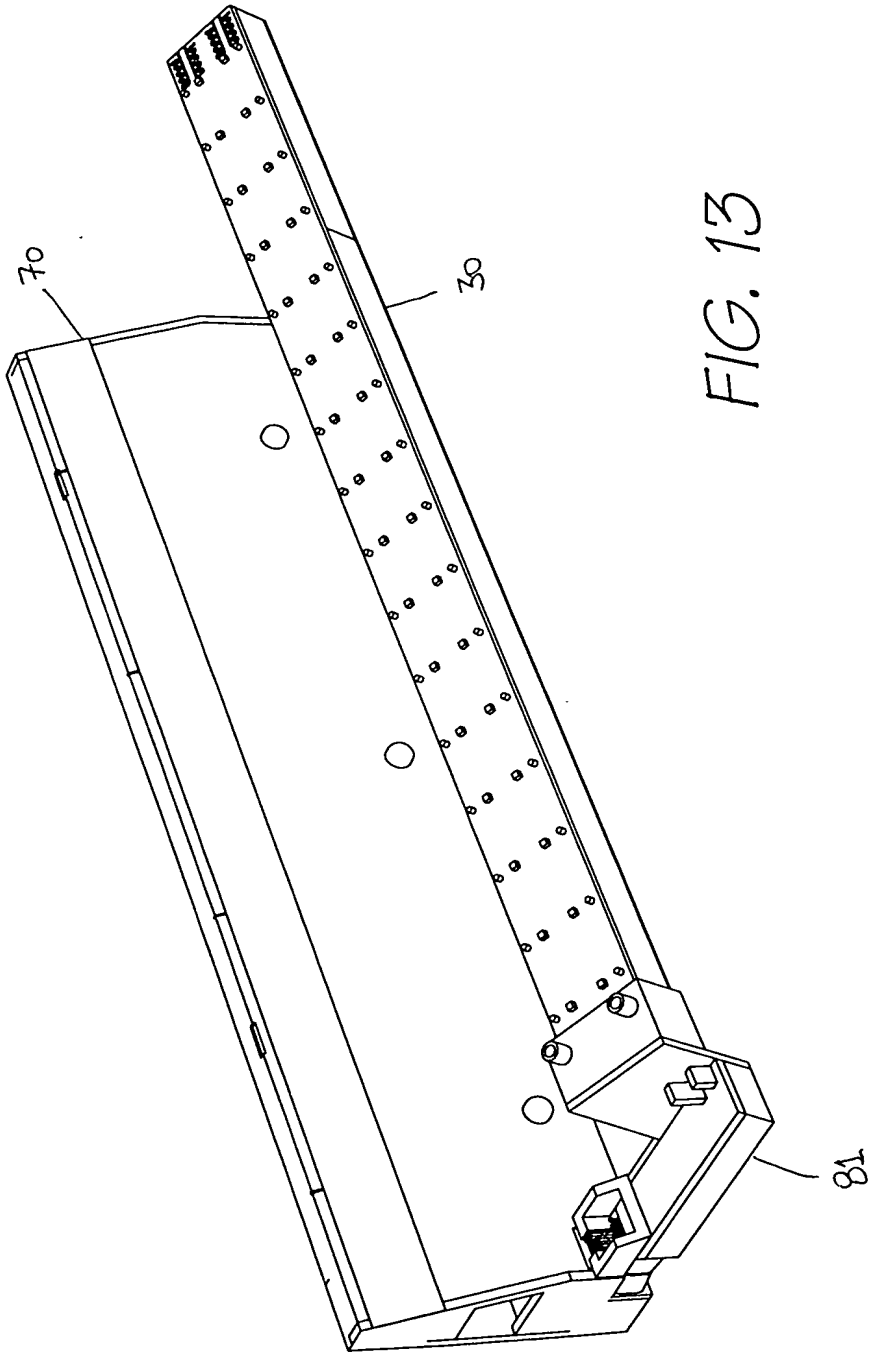


FIG. 13

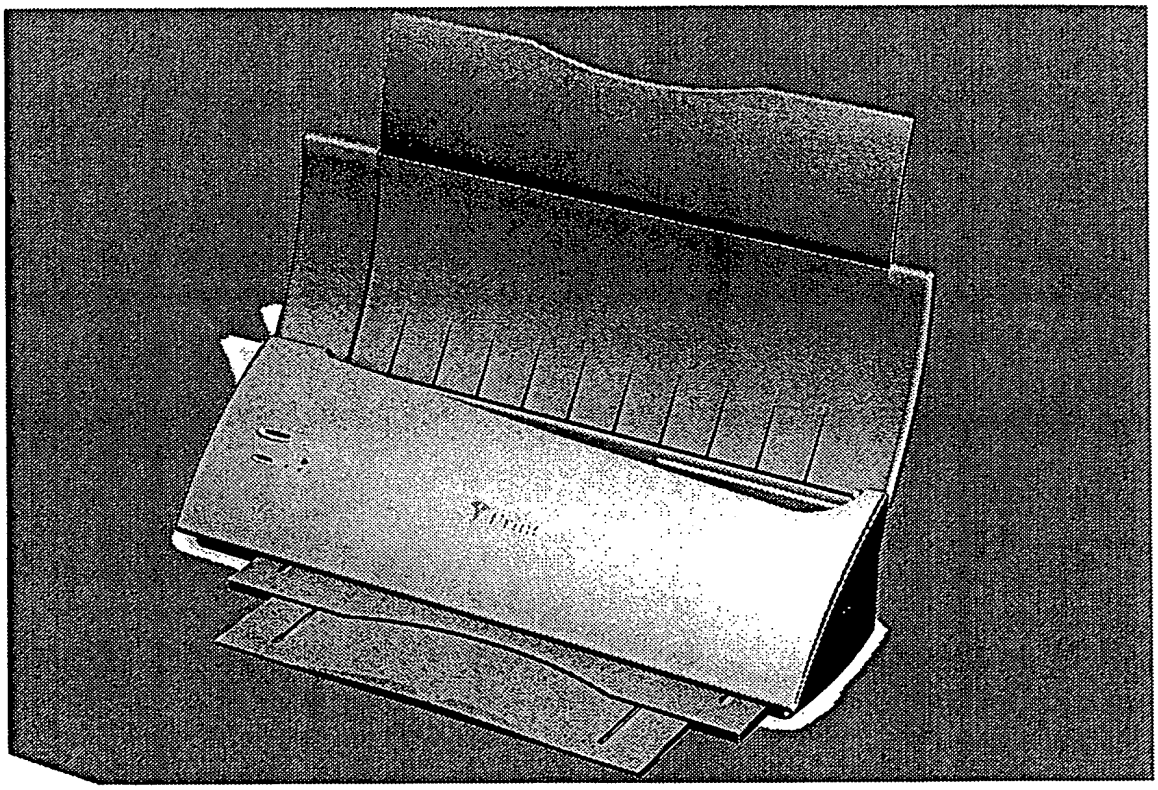
Abstract

A print processor for interconnecting with a computer system and for the production of output pixel data for driving a printhead to produce an output image, comprising: an interface unit for interconnection with the computer system and the receipt of commands therefrom; a bi-level expansion unit interconnected to the interface unit and adapted to receive compressed data for a single color and to expand the compressed data to provide a pixel level representation of the single color data; a pixel expansion unit interconnect to the interface unit and adapted to receive compressed data for the remaining colors of the output image and to decompress the compressed data of the remaining colors to provide a pixel level representation of the remaining color data; compositing means interconnected with the pixel expansion unit and the bi-level expansion unit and adapted to composite the single color data and the remaining color data together to produce image pixel data.

APPENDIX A

iPrint
Product Design Description

draft version 1.0



Silverbrook Research Pty Ltd
393 Darling Street, Balmain
NSW 2041 Australia
Phone: +61 2 9818 6633
Fax: +61 2 9818 6711
Email: info@silverbrook.com.au

Document History

Version	Date	Authors	Details
1.0d	6 November 1998	Paul Lapstun Simon Walmsley Toby King Kia Silverbrook	Initial draft issue.

Contents

1 Introduction

iPrint is a high-performance color printer which combines photographic-quality image reproduction with magazine-quality text reproduction. It prints 30 full-color A4 or Letter pages per minute.

iPrint utilizes an 8" page-width Memjet [17] printhead which produces 1600 dots per inch (dpi) bi-level CMYK (Cyan, Magenta, Yellow, blacK).

iPrint is intended as an entry-level desktop printer.

1.1 Operational Overview

iPrint reproduces black text and graphics directly using bi-level black, and continuous-tone (contone) images and graphics using dithered bi-level CMYK. For practical purposes, iPrint supports a black resolution of 800 dpi, and a contone resolution of 267 pixels per inch (ppi).

iPrint is attached to a workstation or personal computer (PC) via a relatively low-speed (1.5MBytes/s) universal serial bus (USB) connection. iPrint relies on the PC to render each page to the level of contone pixels and black dots. The PC compresses each rendered page to less than 3MB for sub-two-second delivery to the printer. iPrint decompresses and prints the page line by line at the speed of the Memjet printhead. iPrint contains sufficient buffer memory for two compressed pages (6MB), allowing it to print one page while receiving the next, but does not contain sufficient buffer memory for even a single uncompressed page (119MB).

1.2 This Document

This document begins by describing Memjet-based printing in general (Section 2), and the iPrint page delivery architecture which dictates much of iPrint's design (Section 3).

It then goes on to specify the design of the printer hardware (Section 4), the printer control protocol (Section 5), the Memjet printhead (Section 6), the printer controller chip (Section 7), a generic host-based printer driver (Section 8), and a Windows 9x/NT printer driver (Section 9).

The printer hardware section (Section 4) includes estimated manufacturing costs for iPrint and its ink cartridge consumables.

2 Memjet-Based Printing

A Memjet printhead produces 1600 dpi bi-level CMYK. On low-diffusion paper, each ejected drop forms an almost perfectly circular 16μm diameter dot. Dots are easily produced in isolation, allowing dispersed-dot dithering to be exploited to its fullest. Since the Memjet printhead is page-width and operates with a constant paper velocity, the four color planes are printed in perfect registration, allowing ideal dot-on-dot printing. Since there is consequently no spatial interaction between color planes, the same dither matrix is used for each color plane.

A page layout may contain a mixture of images, graphics and text. Continuous-tone (contone) images and graphics are reproduced using a stochastic dispersed-dot dither. Unlike a clustered-dot (or amplitude-modulated) dither, a *dispersed-dot* (or frequency-modulated) dither reproduces high spatial frequencies (i.e. image detail) almost to the limits of the dot resolution, while simultaneously reproducing lower spatial frequencies to their full color depth. A *stochastic* dither matrix is carefully designed to be free of objectionable low-frequency patterns when tiled across the image. As such its size typically exceeds the minimum size required to support a number of intensity levels (i.e. 16×16×8 bits for 257 intensity levels). iPrint uses a dither *volume* of size 64×64×4×8 bits. The volume provides an extra degree of freedom during the design of the dither by allowing a dot to change states multiple times through the intensity range (rather than just once as in a conventional dither matrix).

Human contrast sensitivity peaks at a spatial frequency of about 2 cycles per degree of visual field and then falls off exponentially with an effective upper limit of about 30 cycles per degree [2]. At a normal viewing distance of between 400mm and 300mm, this translates to 100-150 cycles per inch (cpi) on the printed page, or 200-300 samples per inch according to Nyquist's theorem. This places an upper limit of about 300 pixels per inch (ppi) on any contone image printed. Any contone resolution higher than this is mostly wasted, and in fact contributes slightly to color error through the dither.

Black text and graphics are reproduced directly using bi-level black dots. Beyond the perceptual limits discussed above, text resolution up to about 1200 dpi continues to contribute to perceived text sharpness (assuming low-diffusion paper, of course).

3 Page Delivery Architecture

3.1 CONSTRAINTS

USB (Universal Serial Bus) is the standard low-speed peripheral connection on new PCs [5]. The standard high-speed peripheral connection, IEEE 1394, is recommended but unfortunately still optional in the PC 99 specification [6], and so may not be in widespread use when iPrint is first launched. iPrint therefore connects to a personal computer (PC) or workstation via USB, and the speed of the USB connection therefore imposes the most significant constraint on the architecture of the iPrint system. At a sustained printing rate of 30 pages/minute, USB at 1.5MByte/s imposes an average limit of 3MB/page. Since the act of interrupting a Memjet-based printer during the printing of a page produces a visible discontinuity, it is advantageous for the printer to receive the entire page before commencing printing, to eliminate the possibility of buffer underrun. Since the printer can contain only limited buffer memory, i.e. two pages' worth or 6MB, then the 3MB/page limit must be considered absolute.

Figure 1 illustrates the sustained printing rate achievable with double-buffering in the printer.

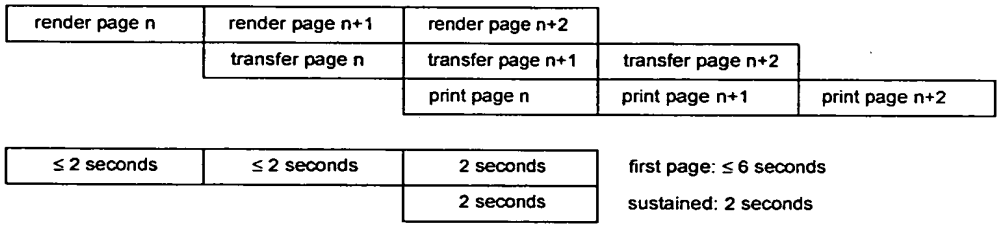


Figure 1. First page and sustained printing rate

Other desktop connection options provide similar bandwidth to USB, and so impose similar constraints on the architecture. These include the parallel port at 2MB/s, and 10Base-T Ethernet at around 1MB/s.

3.2 PAGE RENDERING AND COMPRESSION

Page rendering (or rasterization) can be split between the PC and printer in various ways. Some printers support a full page description language (PDL) such as Postscript, and contain correspondingly sophisticated renderers. Other printers provide special support only for rendering text, to achieve high text resolution. This usually includes support for built-in or downloadable fonts. In each case the use of an embedded renderer reduces the rendering burden on the PC and reduces the amount of data transmitted from the PC to the printer. However, this comes at a price. These printers are more complex than they might be, and are often unable to provide full support for the graphics system of the PC, through which application programs construct, render and print pages. They often fail to exploit the high performance of current PCs, and are unable to leverage projected exponential growth in PC performance.

iPrint relies on the PC to render pages, i.e. contone images and graphics to the pixel level, and black text and graphics to the dot level. iPrint contains only a simple rendering engine which dithers the contone data and combines the results with any foreground bi-level

black text and graphics. This strategy keeps the printer simple, and independent of any page description language or graphics system. It fully exploits the high performance of current PCs. The downside of this strategy is the potentially large amount of data which must be transmitted from the PC to the printer. We consequently use compression to reduce this data to the 3MB/page required to allow a sustained printing rate of 30 pages/minute.

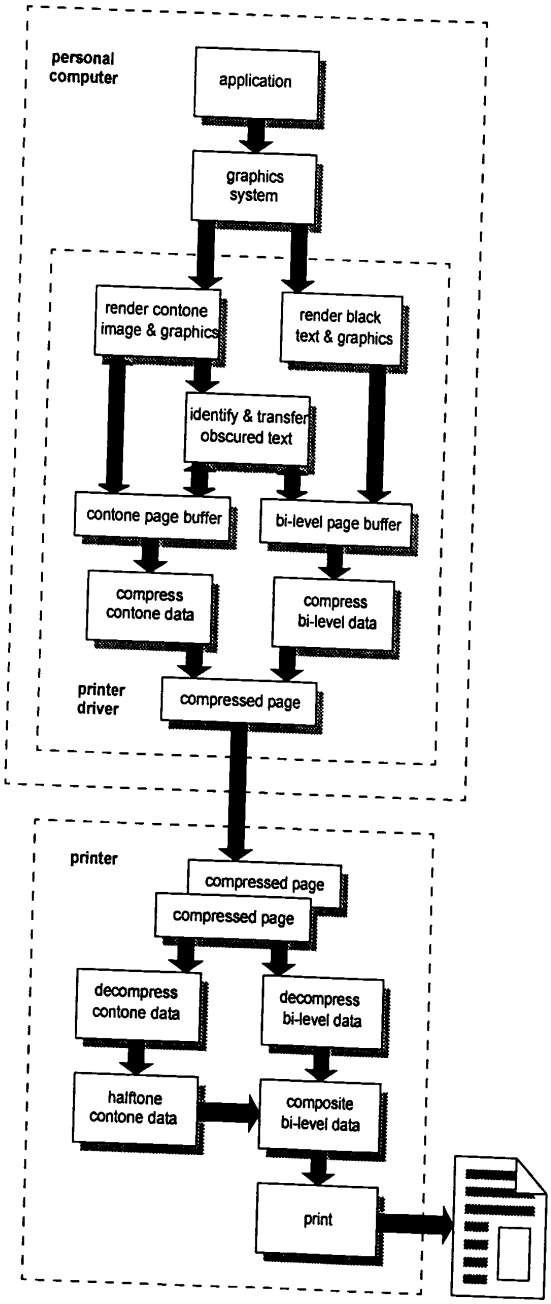


Figure 2. Conceptual data flow from application to printed page

An 8" by 11.7" A4 page has a bi-level CMYK pagesize of 114.3MBytes at 1600 dpi, and a contone CMYK pagesize of 32.1MB at 300 ppi.

We use JPEG compression to compress the contone data. Although JPEG is inherently lossy, for compression ratios of 10:1 or less the loss is usually negligible [22]. To obtain an integral contone to bi-level ratio, and to provide some compression leeway, we choose a contone resolution of 267 ppi. This yields a contone CMYK pagesize of 25.5MB, a corresponding compression ratio of 8.5:1 to fit within the 3MB/page limit, and a contone to bi-level ratio of 1:6 in each dimension.

A full page of black text (and/or graphics) rasterized at printer resolution (1600 dpi) yields a bi-level image of 28.6MB. Since rasterizing text at 1600 dpi places a heavy burden on the PC for a small gain, we choose to rasterize text at a fully acceptable 800 dpi. This yields a bi-level image of 7.1MB, requiring a lossless compression ratio of less than 2.5:1 to fit within the 3MB/page limit. We achieve this with a two-dimensional compression scheme adapted from Group 4 Facsimile.

As long as the image and text regions of a page are non-overlapping, any combination of the two fits within the 3MB limit. If text lies on top of a background image, then the worst case is a compressed pagesize approaching 6MB (depending on the actual text compression ratio). This fits within the printer's page buffer memory, but prevents double-buffering of pages in the printer, thereby reducing the printer's page rate by two-thirds, i.e. to 15 pages/minute.

3.3 Page Expansion and Printing

As described above, the PC renders contone images and graphics to the pixel level, and black text and graphics to the dot level. These are compressed by different means and transmitted together to the printer.

The printer contains two 3MB page buffers - one for the page being received from the PC, and one for the page being printed. The printer expands the compressed page as it is being printed. This expansion consists of decompressing the 267 ppi contone CMYK image data, halftoning the resulting contone pixels to 1600 dpi bi-level CMYK dots, decompressing the 800 dpi bi-level black text data, and compositing the resulting bi-level black text dots *over* the corresponding bi-level CMYK image dots.

The conceptual data flow from the application to the printed page is illustrated in Figure 2.

4 Printer Hardware

Because of the simplicity of the pagewidth Memjet printhead, iPrint is very compact. It measures just 270mm wide × 85mm deep × 77mm high when closed.

The cover opens to form part of the paper tray, with the second part hinged within. The paper exit tray extends from the front.

The front panel contains the user interface - the power button and power indicator LED, the paper feed button, and the out-of-paper and ink low LEDs.

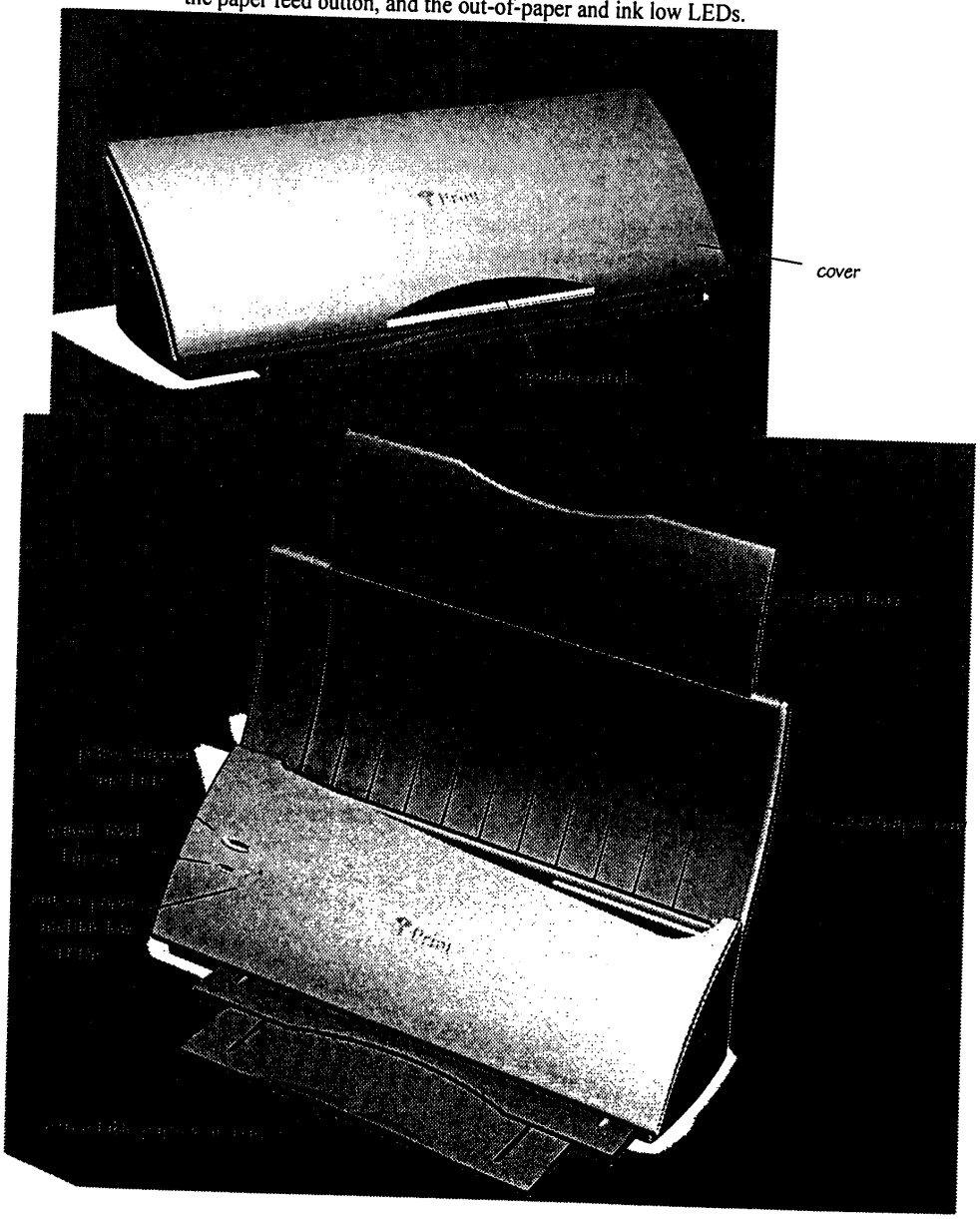


Figure 3. iPrint closed, and open with paper trays extended

4.1 PAPER PATH

iPrint uses a standard paper transport mechanism in which a single stepper motor drives both the sheet feed and the paper transport. When running in the forward direction the stepper motor drives the paper drive roller and the pinch wheels at the start and end of the active paper path, respectively. When reversed, the stepper motor drives the sheet feed roller which grabs the topmost sheet from the sheet feeder and transports it the short distance to the paper drive roller where it is detected by the mechanical media sensor.

The paper centering sliders ensure that the paper is centered. This ensures that a single centered media sensor detects the sheet, and also ensures that sheets wider than the printhead are printed with balanced margins.

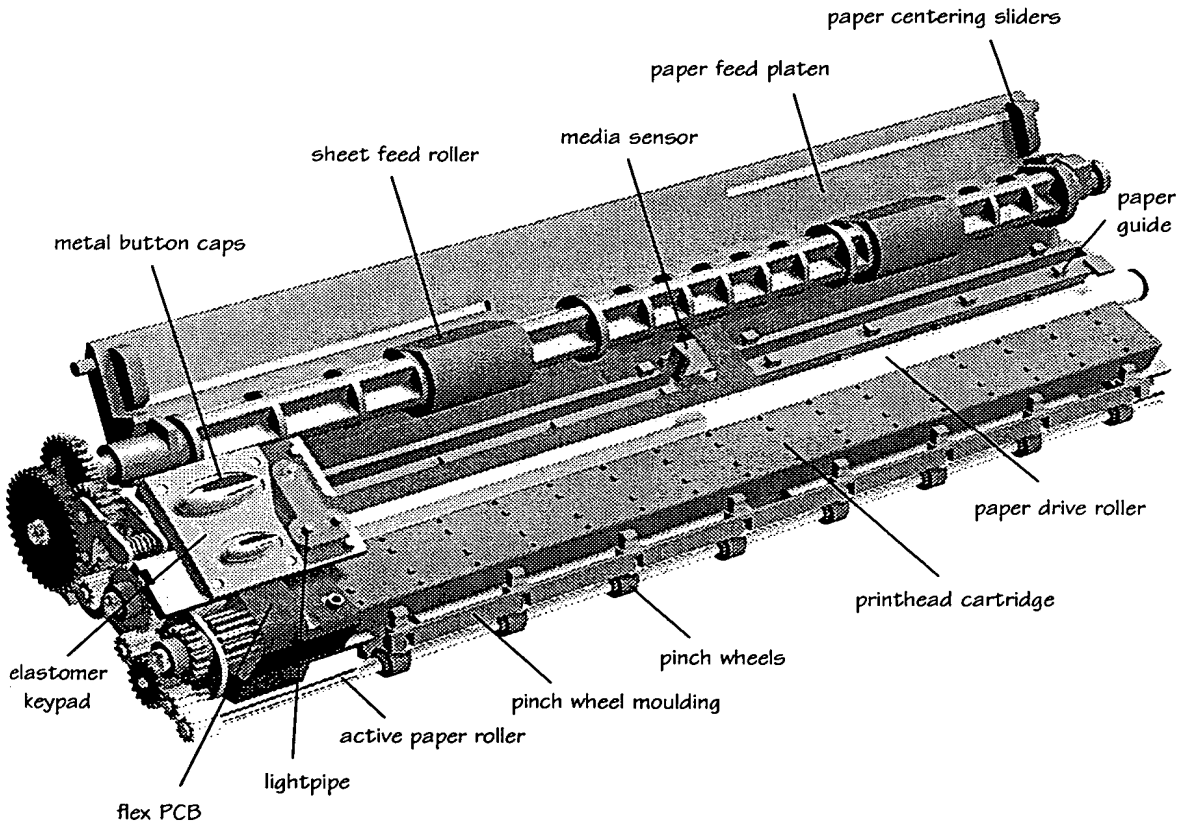


Figure 4. Exposed paper transport mechanism

4.1.1 Memjet Printhead

The replaceable Memjet printhead cartridge is shown in Figure 4. This represents one of the four possible ways to deploy the printhead in conjunction with the ink cartridge in a product such as iPrint:

- permanent printhead, replaceable ink cartridge
- separate replaceable printhead and ink cartridges
- refillable combined printhead and ink cartridge
- disposable combined printhead and ink cartridge

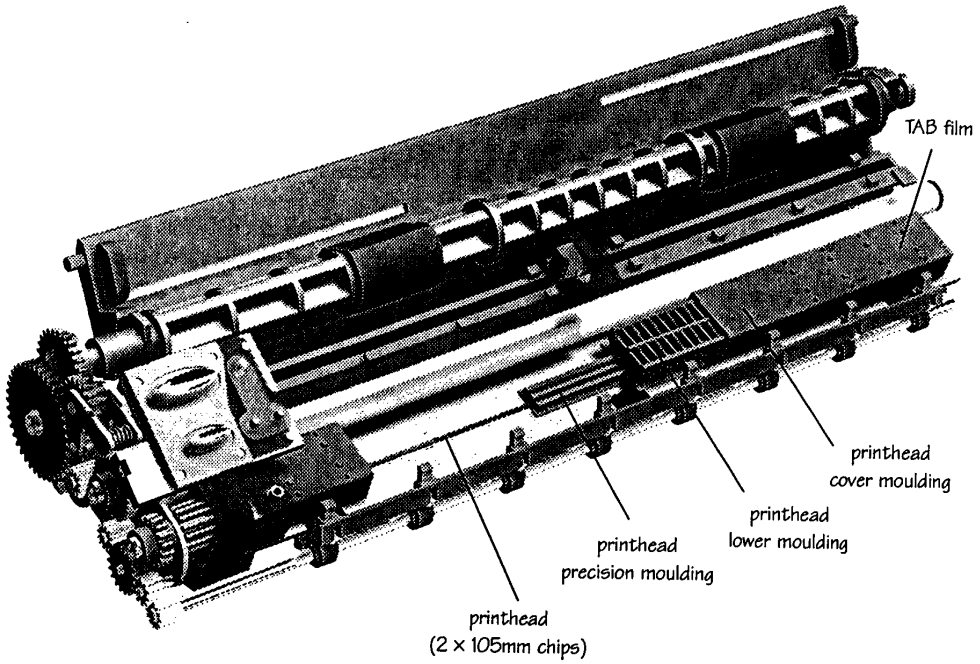


Figure 5. Printhead cartridge cutaway

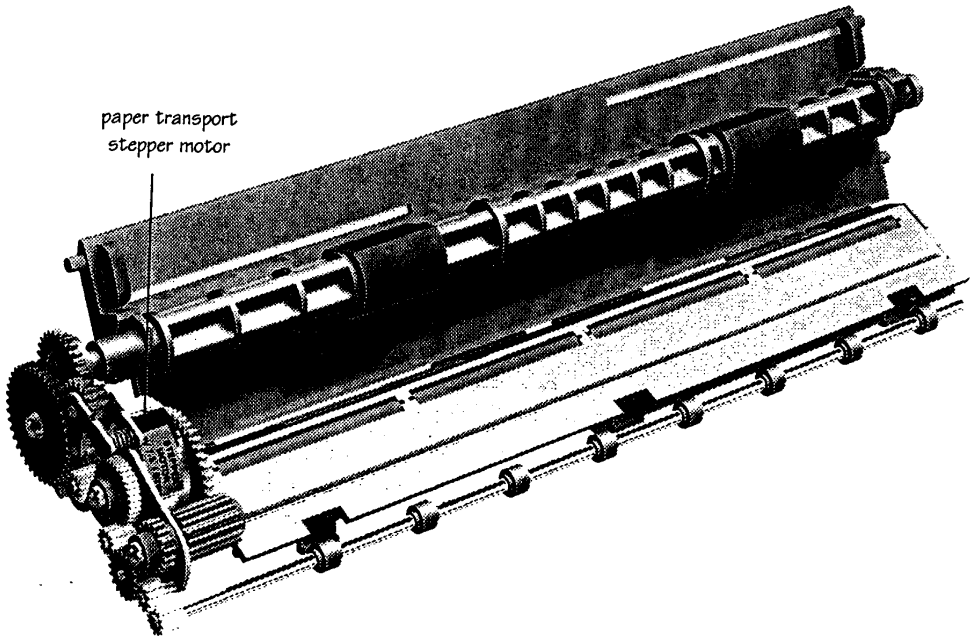


Figure 6. Fully-exposed paper path

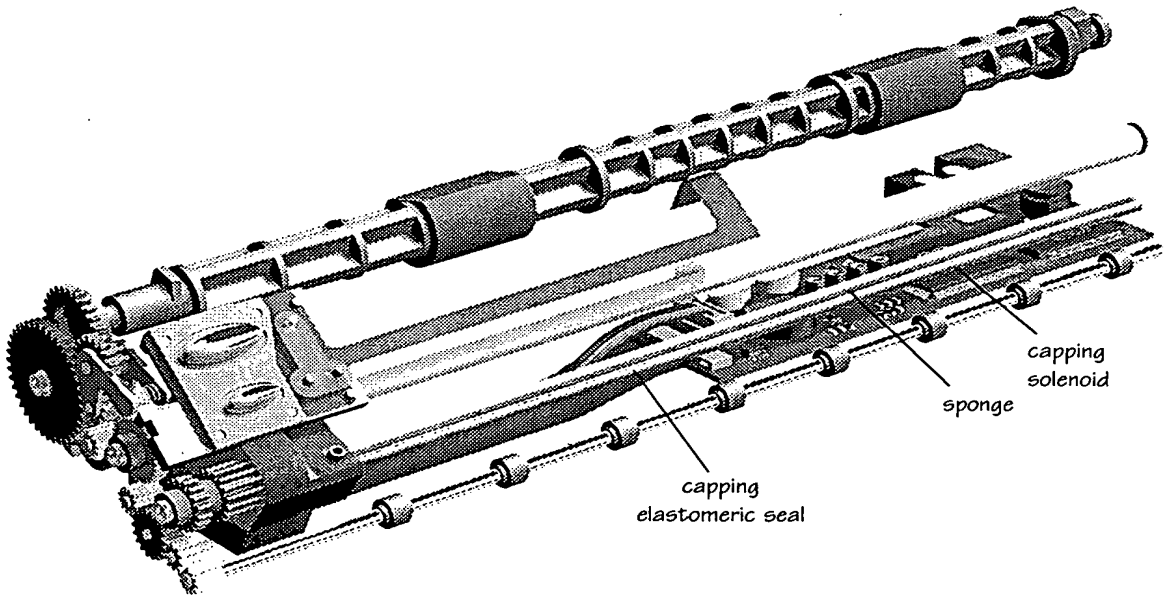


Figure 7. Printhead capping mechanism and sponge

When the printhead is not in use, the elastomeric seal is in contact with the printhead precision moulding, capping the printhead. When the printhead is in use, the capping solenoid, which runs the full length of the printhead, is activated and holds the capping seal away from the printhead, away from the paper path.

4.2 PRINTER CONTROLLER

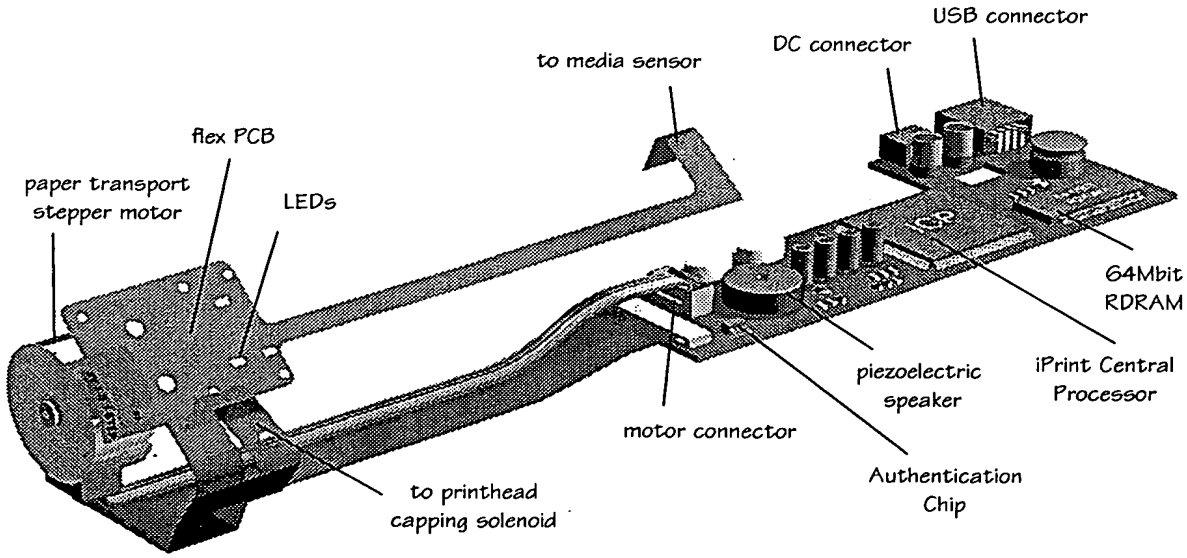


Figure 8. Printer controller PCB and flex PCB

The printer controller consists of a small PCB with only a few components - a 64Mbit RDRAM, the iPrint Central Processor (ICP) chip, piezoelectric speaker for notifying the user of error conditions, the Authentication Chip, an external 3V DC power connection, an external USB connection, a connection to the paper transport stepper motor, and the flex PCB which connects to the media sensor, LEDs, buttons, and the printhead capping solenoid.

4.3 Ink Cartridge and Ink Path

There are two versions of the ink cartridge - one small, one large. Both fit in the same ink cartridge slot at the back of the iPrint unit, shown in Figure 10. The ink path from the ink cartridge to the printhead is shown in Figure 11.

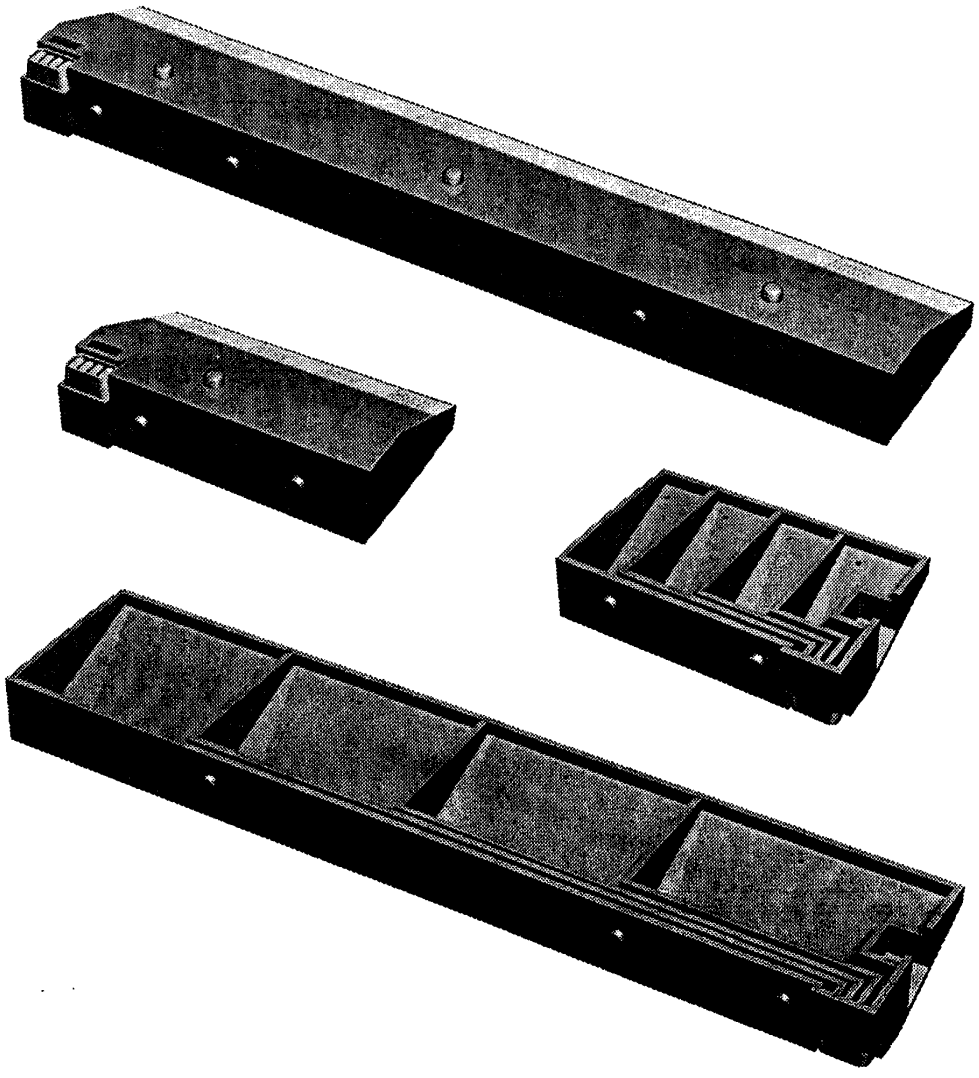


Figure 9. Large and small ink cartridge designs

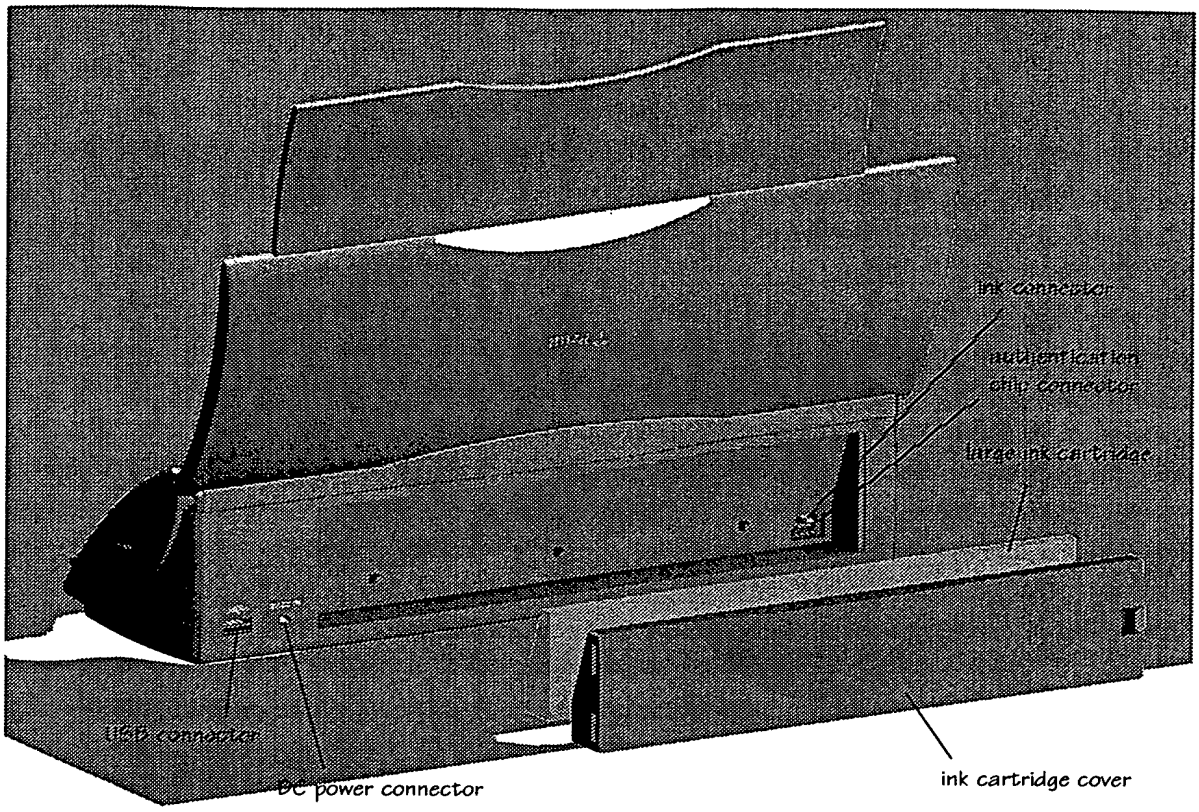


Figure 10. Rear view with large ink cartridge and ink cartridge slot

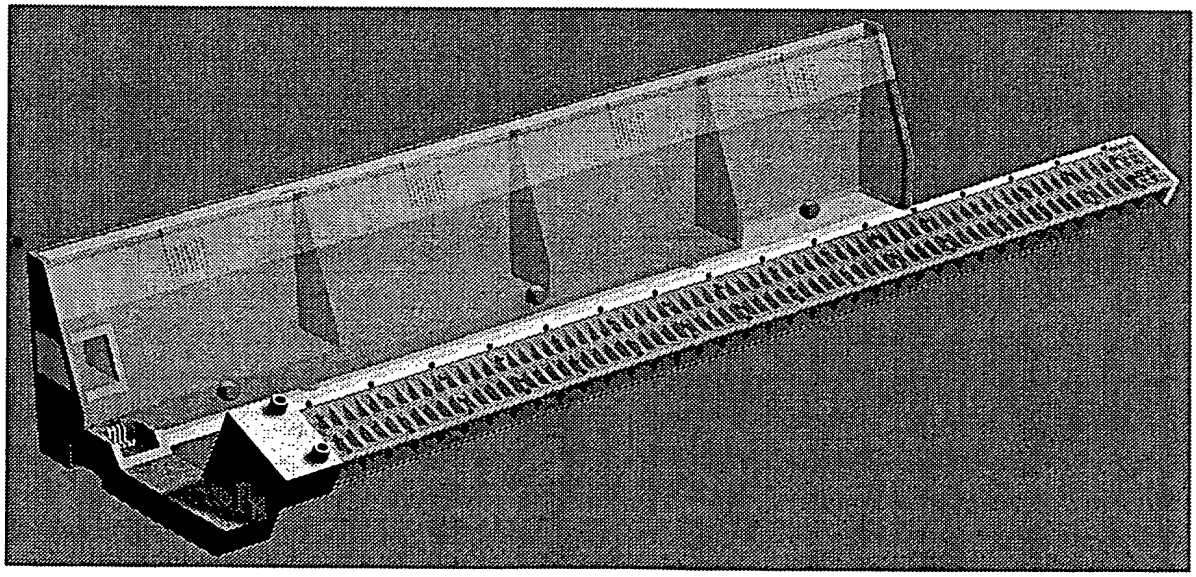


Figure 11. Ink path from large ink cartridge to printhead

4.4 MANUFACTURING COST

The manufacturing cost for iPrint is given in Table 1. The manufacturing costs for the small and large ink cartridges are given in Table 2 and Table 3 respectively.

Table 1. iPrint manufacturing cost

item	no.	unit cost	cost
Back Plate moulding	1	$0.200	$0.20
Base moulding	1	$0.200	$0.20
Metal foil	2	$0.010	$0.02
Screws	5	$0.005	$0.03
Internal Paper tray door	1	$0.100	$0.10
Upper Paper tray door	1	$0.100	$0.10
Paper tray door	1	$0.100	$0.10
Metal shim	1	$0.010	$0.01
Printed logo	1	$0.010	$0.01
Feed moulding	1	$0.050	$0.05
Feed Slider mouldings	2	$0.010	$0.02
Feed Slider Caps	2	$0.010	$0.02
Feed Washer moulding	1	$0.005	$0.01
Retaining clips	2	$0.005	$0.01
Chassis metalwork	1	$1.000	$1.00
Screws	2	$0.005	$0.01
Button Support moulding	1	$0.010	$0.01
Elastomeric Keypad (two buttons)	1	$0.010	$0.01
Metal plated moulded button caps	2	$0.005	$0.01
Chassis Paper Guide moulding	1	$0.300	$0.30
Paper centering rack gears	2	$0.020	$0.04
Paper centering pinion gear	1	$0.005	$0.01
Chassis Pinch Wheel Retainer moulding	1	$0.020	$0.02
Lightpipe moulding	1	$0.010	$0.01
Platen moulding	1	$0.040	$0.04
Plastic foil	2	$0.005	$0.01
Platen Rollers (metal turnings)	4	$0.020	$0.08
Platen Solenoid & contacts	1	$0.100	$0.10
Paper exit tray retainer blocks	2	$0.010	$0.02
Screws	4	$0.005	$0.02
Paper exit extender tray	1	$0.020	$0.02
Retaining clips	2	$0.005	$0.01
Rubber feet	4	$0.005	$0.02
Label	1	$0.010	$0.01

Table 1. iPrint manufacturing cost

item	no.	unit cost	cost
Rigid PCB (double sided)	1	$1.000	$1.00
USB connector	1	$0.100	$0.10
DC connector	1	$0.050	$0.05
iPrint Central Processor ASIC (0.18 micron lithography)	1	$4.000	$4.00
64 Mbit DRAM (year 2001) (0.13 micron lithography)	1	$4.000	$4.00
Voltage regulator	1	$0.100	$0.10
Surge protector	1	$0.020	$0.02
Surface mount resistors	14	$0.005	$0.07
Ceramic decoupling capacitors	4	$0.005	$0.02
Tantalum decoupling capacitors	4	$0.010	$0.04
Electrolytic capacitors	4	$0.020	$0.08
Motor drive transistors	4	$0.010	$0.04
Motor connector	1	$0.050	$0.05
Flex PCB connector	1	$0.050	$0.05
Flex PCB	1	$1.000	$1.00
Authentication chip connector	1	$0.050	$0.05
Paper Sensor	1	$0.200	$0.20
Printhead cartridge cover moulding	1	$0.050	$0.05
Printhead cartridge mounting extrusion	1	$0.010	$0.01
Printhead cartridge base moulding	1	$0.050	$0.05
Printhead cartridge precision moulding	1	$0.100	$0.10
Printhead cartridge seal moulding	1	$0.020	$0.02
Printhead chip	2	$3.000	$6.00
Printhead cartridge filter	1	$0.100	$0.10
TAB film	1	$1.000	$1.00
Copper bus bars (gold plated)	2	$0.100	$0.20
Bus bar contact strips	2	$0.020	$0.04
Stepper motor & leads	1	$0.500	$0.50
Motor chassis	1	$0.020	$0.02
Screws	3	$0.005	$0.02
Paper feed rod	1	$0.030	$0.03
Paper feed rod grippers	2	$0.010	$0.02
Paper feed gear	1	$0.010	$0.01
Paper feed retainer mouldings	2	$0.010	$0.02
Large drive gear	1	$0.010	$0.01
Planetary gear cage moulding (nylon)	1	$0.010	$0.01
Planetary gear	1	$0.005	$0.01

Table 1. iPrint manufacturing cost

item	no.	unit cost	cost
Double gear	1	$0.005	$0.01
Reversing gear	1	$0.005	$0.01
Reversing gear arm moulding	1	$0.005	$0.01
Helical steel spring	1	$0.005	$0.01
Reduction gear (nylon)	1	$0.005	$0.01
Small drive gear	1	$0.005	$0.01
Intermediate gear	1	$0.005	$0.01
Gear moulding & screw	1	$0.010	$0.01
Small transmission gear	1	$0.005	$0.01
Pinch roller moulding & integrated gear	1	$0.020	$0.02
Paper drive roller	1	$0.200	$0.20
Paper drive roller gear	1	$0.005	$0.01
Paper drive roller end cap moulding	1	$0.005	$0.01
Pinch 'spike' wheels	8	$0.010	$0.08
PCB pick and place	1	$1.000	$1.00
Automated PCB test	1	$0.500	$0.50
PCB rework	0.1	$2.000	$0.20
PCB yield loss	0.01	$10.000	$0.10
Automated assembly	1	$2.000	$2.00
Manual assembly	1	$2.000	$2.00
Automated testing	1	$1.000	$1.00
Rework	0.01	$10.000	$0.10
Packing	1	$0.100	$0.10
Wrapper	1	$0.050	$0.05
Instruction book	1	$0.200	$0.20
Cardboard box	1	$0.100	$0.10
Total			$29.41

Table 2. Small ink cartridge manufacturing cost

item	no.	unit cost	cost
Upper moulding	1	$0.020	$0.02
Lower moulding	1	$0.015	$0.02
Elastomeric seal moulding	1	$0.010	$0.01
Nozzle moulding	1	$0.020	$0.02
Authentication chip	1	$0.100	$0.10
Ink sponges	4	$0.020	$0.08
Cyan ink (liter)	0.025	$4.000	$0.10
Magenta ink (liter)	0.025	$4.000	$0.10
Yellow ink (liter)	0.025	$4.000	$0.10

Table 2. Small ink cartridge manufacturing cost

item	no.	unit cost	cost
Black ink (liter)	0.1	$3.000	$0.30
Front pressure sensitive label	1	$0.010	$0.01
Back pressure sensitive label	1	$0.010	$0.01
Seal strip	1	$0.010	$0.01
Airtight plastic bag	1	$0.010	$0.01
Instruction sheet	1	$0.010	$0.01
Cardboard box	1	$0.020	$0.02
Automated assembly	1	$0.050	$0.05
Total			$0.97

Table 3. Large ink cartridge manufacturing cost

item	no.	unit cost	cost
Upper moulding	1	$0.040	$0.04
Lower moulding	1	$0.030	$0.03
Elastomeric seal moulding	1	$0.020	$0.02
Nozzle moulding	1	$0.020	$0.02
Authentication chip	1	$0.100	$0.10
Ink sponges	4	$0.050	$0.20
Cyan ink (liter)	0.1	$4.000	$0.40
Magenta ink (liter)	0.1	$4.000	$0.40
Yellow ink (liter)	0.1	$4.000	$0.40
Black ink (liter)	0.3	$3.000	$0.90
Front pressure sensitive label	1	$0.020	$0.02
Back pressure sensitive label	1	$0.020	$0.02
Seal strip	1	$0.010	$0.01
Airtight plastic bag	1	$0.020	$0.02
Instruction sheet	1	$0.010	$0.01
Cardboard box	1	$0.040	$0.04
Automated assembly	1	$0.050	$0.05
Total			$2.68

5 Printer Control Protocol

This section describes the printer control protocol used between a host and iPrint. It includes control and status handling as well as the actual page description.

5.1 CONTROL AND STATUS

The USB device class definition for printers [21] provides for emulation of both unidirectional and bidirectional IEEE 1284 parallel ports [3]. At its most basic level, this allows the host to determine printer capabilities (via GET_DEVICE_ID), obtain printer status (via GET_PORT_STATUS), and reset the printer (via SOFT_RESET). Centronics/IEEE 1284 printer status fields are described below.

Table 4. Centronics/IEEE 1284 printer status

field	description
Select	The printer is selected and available for data transfer.
Paper Empty	A paper empty condition exists in the printer.
Fault	A fault condition exists in the printer (includes Paper Empty and not Select).

Personal computer printing subsystems typically provide some level of IEEE 1284 support. Compatibility with IEEE 1284 in a printer therefore simplifies the development of the corresponding printer driver. The USB device class definition for printers seeks to leverage this same compatibility.

iPrint supports no control protocol beyond the USB device class definition for printers. Note that, if a higher-level control protocol were defined, then conditions such as out-of-ink could also be reported to the user (rather than just via the printer's out-of-ink LED).

iPrint receives page descriptions as raw transfers, i.e. not encapsulated in any higher-level control protocol.

5.2 PAGE DESCRIPTION

iPrint reproduces black at full dot resolution (1600 dpi), but reproduces contone color at a somewhat lower resolution using halftoning. The page description is therefore divided into a black layer and a contone layer. The black layer is defined to composite *over* the contone layer.

The black layer consists of a bitmap containing a 1-bit *opacity* for each pixel. This black layer *matte* has a resolution which is an integer factor of the printer's dot resolution. The highest supported resolution is 1600 dpi, i.e. the printer's full dot resolution.

The contone layer consists of a bitmap containing a 32-bit CMYK *color* for each pixel. This contone image has a resolution which is an integer factor of the printer's dot resolution. The highest supported resolution is 267 ppi, i.e. one-sixth the printer's dot resolution.

The contone resolution is also typically an integer factor of the black resolution, to simplify calculations in the printer driver. This is not a requirement, however.

The black layer and the contone layer are both in compressed form for efficient transmission over the low-speed USB connection to the printer.

Table 5 shows the format of the page description expected by iPrint.

Table 5. Page description format

field	format	description
signature	16-bit integer	Page description format signature.
version	16-bit integer	Page description format version number.
structure size	16-bit integer	Size of fixed-size part of page description.
target resolution (dpi)	16-bit integer	Resolution of target page. This is always 1600 for iPrint.
target page width	16-bit integer	Width of target page, in dots.
target page height	16-bit integer	Height of target page, in dots.
black scale factor	16-bit integer	Scale factor from black resolution to target resolution (must be 2 or greater).
black page width	16-bit integer	Width of black page, in black pixels.
black page height	16-bit integer	Height of black page, in black pixels.
black page data size	32-bit integer	Size of black page data, in bytes.
contone scale factor	16-bit integer	Scale factor from contone resolution to target resolution (must be 6 or greater).
contone page width	16-bit integer	Width of contone page, in contone pixels.
contone page height	16-bit integer	Height of contone page, in contone pixels.
contone page data size	32-bit integer	Size of contone page data, in bytes.
black page data	EDRL bytestream	Compressed bi-level black page data.
contone page data	JPEG bytestream	Compressed contone CMYK page data.

Each page description is complete and self-contained. There is no data transmitted to the printer separately from the page description to which the page description refers.

The page description contains a signature and version which allow the printer to identify the page description format. If the signature and/or version are missing or incompatible with the printer, then the printer can reject the page.

The page description defines the resolution and size of the target page. The black and contone layers are clipped to the target page if necessary. This happens whenever the black or contone scale factors are not factors of the target page width or height.

The black layer parameters define the pixel size of the black layer, its integer scale factor to the target resolution, and the size of its compressed page data. The variable-size black page data follows the fixed-size parts of the page description.

The contone layer parameters define the pixel size of the contone layer, its integer scale factor to the target resolution, and the size of its compressed page data. The variable-size contone page data follows the variable-size black page data.

All integers in the page description are stored in big-endian byte order.

The variable-size black page data and the variable-size contone page data are aligned to 8-byte boundaries. The size of the required padding is included in the size of the fixed-size part of the page description structure and the variable-size black data.

The entire page description has a target size of less than 3MB, and a maximum size of 6MB, in accordance with page buffer memory in the printer.

The following sections describe the format of the compressed black layer and the compressed contone layer.

5.2.1 Bi-level Black Layer Compression

5.2.1.1 *Group 3 and 4 Facsimile Compression*

The Group 3 Facsimile compression algorithm [1] losslessly compresses bi-level data for transmission over slow and noisy telephone lines. The bi-level data represents scanned black text and graphics on a white background, and the algorithm is tuned for this class of images (it is explicitly not tuned, for example, for *halftoned* bi-level images). The *1D* Group 3 algorithm runlength-encodes each scanline and then Huffman-encodes the resulting runlengths. Runlengths in the range 0 to 63 are coded with *terminating* codes. Runlengths in the range 64 to 2623 are coded with *make-up* codes, each representing a multiple of 64, followed by a terminating code. Runlengths exceeding 2623 are coded with multiple make-up codes followed by a terminating code. The Huffman tables are fixed, but are separately tuned for black and white runs (except for make-up codes above 1728, which are common). When possible, the *2D* Group 3 algorithm encodes a scanline as a set of short edge deltas (0, ±1, ±2, ±3) with reference to the previous scanline. The delta symbols are entropy-encoded (so that the zero delta symbol is only one bit long etc.) Edges within a 2D-encoded line which can't be delta-encoded are runlength-encoded, and are identified by a prefix. 1D- and 2D-encoded lines are marked differently. 1D-encoded lines are generated at regular intervals, whether actually required or not, to ensure that the decoder can recover from line noise with minimal image degradation. 2D Group 3 achieves compression ratios of up to 6:1 [19].

The Group 4 Facsimile algorithm [1] losslessly compresses bi-level data for transmission over *error-free* communications lines (i.e. the lines are truly error-free, or error-correction is done at a lower protocol level). The Group 4 algorithm is based on the 2D Group 3 algorithm, with the essential modification that since transmission is assumed to be error-free, 1D-encoded lines are no longer generated at regular intervals as an aid to error-recovery. Group 4 achieves compression ratios ranging from 20:1 to 60:1 for the CCITT set of test images [19].

The design goals and performance of the Group 4 compression algorithm qualify it as a compression algorithm for the bi-level black layer. However, its Huffman tables are tuned to a lower scanning resolution (100-400 dpi), and it encodes runlengths exceeding 2623 awkwardly. At 800 dpi, our maximum runlength is currently 6400. Although a Group 4 decoder core might be available for use in the printer controller chip (Section 7), it might not handle runlengths exceeding those normally encountered in 400 dpi facsimile applications, and so would require modification.

Since most of the benefit of Group 4 comes from the delta-encoding, a simpler algorithm based on delta-encoding alone is likely to meet our requirements. This approach is described in detail below.

5.2.1.2 *Bi-Level Edge Delta and Runlength (EDRL) Compression Format*

The *edge delta and runlength* (EDRL) compression format is based loosely on the Group 4 compression format and its precursors [1,23].

EDRL uses three kinds of symbols, appropriately entropy-coded. These are *create edge*, *kill edge*, and *edge delta*. Each line is coded with reference to its predecessor. The predecessor of the first line is defined to a line of white. Each line is defined to start off white. If

a line actually starts of black (the less likely situation), then it must define a black edge at offset zero. Each line must define an edge at its left-hand end, i.e. at offset *page width*.

An edge can be coded with reference to an edge in the previous line if there is an edge within the maximum delta range with the same sense (white-to-black or black-to-white). This uses one of the *edge delta* codes. The likelier or shorter deltas have the shorter codes. The maximum delta range (±2) is chosen to match

Table 6. Edge delta distribution for 10 point Times at 800 dpi

|delta|	probability
0	65%
1	23%
2	7%
≥ 3	5%

An edge can also be coded using the length of the run from the previous edge in the same line. This uses one of the *create edge* codes for short (7-bit) and long (13-bit) runlengths. For simplicity, and unlike Group 4, runlengths are not entropy-coded. In order to keep edge deltas implicitly synchronised with edges in the previous line, each unused edge in the previous line is 'killed' when passed in the current line. This uses the *kill edge* code. The *end-of-page* code signals the end of the page to the decoder.

Note that 7-bit and 13-bit runlenghts are specifically chosen to support 800 dpi A4/Letter pages. Longer runlengths could be supported without significant impact on compression performance. For example, if supporting 1600 dpi compression, the runlengths should be at least 8-bit and 14-bit respectively. A general-purpose choice might be 8-bit and 16-bit, thus supporting up to 40" wide 1600 dpi pages.

The full set of codes is defined in Table 7. Note that there is no *end-of-line* code. The decoder uses the *page width* to detect the end of the line. The lengths of the codes are ordered by the relative probabilities of the codes' occurrence.

Table 7. EDRL codewords

code	encoding	suffix	description
Δ0	1	-	move corresponding edge
Δ+1	010	-	"
Δ−1	011	-	"
Δ+2	00010	-	"
Δ−2	00011	-	"
kill edge	0010	-	kill corresponding edge
create near edge	0011	7-bit RL	create edge from short runlength (RL)
create far edge	00001	13-bit RL	create edge from long runlength (RL)
end-of-page (EOP)	000001	-	end-of-page marker

Figure 12 shows a simple encoding example. Note that the common situation of an all-white line following another all-white line is encoded using a single bit (Δ0), and an all-black line following another all-black line is encoded using two bits (Δ0, Δ0).

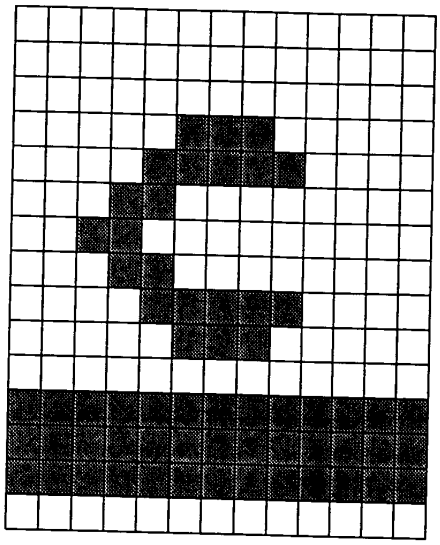


Figure 12. EDRL encoding example

Note that the foregoing describes the compression *format*, not the compression algorithm *per se*. A variety of equivalent encodings can be produced for the same image, some more compact than others. For example, a pure runlength encoding conforms to the compression format. The goal of the compression algorithm is to discover a good, if not the best, encoding for a given image.

The following is a simple algorithm for producing the EDRL encoding of a line with reference to its predecessor.

```
#define SHORT_RUN_PRECISION7  // precision of short run
#define LONG_RUN_PRECISION13  // precision of long run

EDRL_CompressLine
(
   Byte prevLine[],           // previous (reference) bi-level line
   Byte currLine[],           // current (coding) bi-level line
   int lineLen,               // line length
   BITSTREAM s                // output (compressed) bitstream
)
   int prevEdge = 0           // current edge offset in previous line
   int currEdge = 0           // current edge offset in current line
   int codedEdge = currEdge   // most recent coded (output) edge
   int prevColor = 0          // current color in previous line (0 = white)
   int currColor = 0          // current color in current line
   int prevRun                // current run in previous line
   int currRun                // current run in current line
   bool bUpdatePrevEdge = true// force first edge update
   bool bUpdateCurrEdge = true// force first edge update

   while (codedEdge < lineLen)
```

```
// possibly update current edge in previous line
if (bUpdatePrevEdge)
   if (prevEdge < lineLen)
      prevRun = GetRun(prevLine, prevEdge, lineLen, prevColor)
   else
      prevRun = 0
   prevEdge += prevRun
   prevColor = !prevColor
   bUpdatePrevEdge = false

// possibly update current edge in current line
if (bUpdateCurrEdge)
   if (currEdge < lineLen)
      currRun = GetRun(currLine, currEdge, lineLen, currColor)
   else
      currRun = 0
   currEdge += currRun
   currColor = !currColor
   bUpdateCurrEdge = false

// output delta whenever possible, i.e. when
// edge senses match, and delta is small enough
if (prevColor == currColor)
   delta = currEdge - prevEdge
   if (abs(delta) <= MAX_DELTA)
      PutCode(s, EDGE_DELTA0 + delta)
      codedEdge = currEdge
      bUpdatePrevEdge = true
      bUpdateCurrEdge = true
      continue

// kill unmatched edge in previous line
if (prevEdge <= currEdge)
   PutCode(s, KILL_EDGE)
   bUpdatePrevEdge = true

// create unmatched edge in current line
if (currEdge <= prevEdge)
   PutCode(s, CREATE_EDGE)
   if (currRun < 128)
      PutCode(s, CREATE_NEAR_EDGE)
      PutBits(currRun, SHORT_RUN_PRECISION)
   else
      PutCode(s, CREATE_FAR_EDGE)
      PutBits(currRun, LONG_RUN_PRECISION)
   codedEdge = currEdge
   bUpdateCurrEdge = true
```

Figure 13. Algorithm for EDRL-compressing a bi-level line

Note that the algorithm is blind to *actual* edge continuity between lines, and may in fact match the "wrong" edges between two lines. Happily the compression format has nothing to say about this, since it decodes correctly, and it is difficult for a "wrong" match to have a detrimental effect on the compression ratio.

For completeness the corresponding decompression algorithm is given below. It forms the core of the EDRL Expander unit in the printer controller chip (Section 7).

```
EDRL_DecompressLine
(
```

```
    BITSTREAM s,                // input (compressed) bitstream
    Byte prevLine[],            // previous (reference) bi-level line
    Byte currLine[],            // current (coding) bi-level line
    int lineLen                 // line length
)
    int prevEdge = 0            // current edge offset in previous line
    int currEdge = 0            // current edge offset in current line
    int prevColor = 0           // current color in previous line (0 = white)
    int currColor = 0           // current color in current line

    while (currEdge < lineLen)

        code = GetCode(s)
        switch (code)
            case EDGE_DELTA_MINUS2:
            case EDGE_DELTA_MINUS1:
            case EDGE_DELTA_0:
            case EDGE_DELTA_PLUS1:
            case EDGE_DELTA_PLUS2:
                // create edge from delta
                int delta = code - EDGE_DELTA_0
                int run = prevEdge + delta - currEdge
                FillBitRun(currLine, currEdge, currColor, run)
                currEdge += run
                currColor = !currColor
                prevEdge += GetRun(prevLine, prevEdge, lineLen, prevColor)
                prevColor = !prevColor

            case KILL_EDGE:
                // discard unused reference edge
                prevEdge += GetRun(prevLine, prevEdge, lineLen, prevColor)
                prevColor = !prevColor

            case CREATE_NEAR_EDGE:
            case CREATE_FAR_EDGE:
                // create edge explicitly
                int run
                if (code == CREATE_NEAR_EDGE)
                    run = GetBits(s, SHORT_RUN_PRECISION)
                else
                    run = GetBits(s, LONG_RUN_PRECISION)
                FillBitRun(currLine, currEdge, currColor, run)
                currColor = !currColor
                currEdge += run
```

Figure 14. Algorithm for decompressing an EDRL-compressed bi-level line

5.2.1.3 *EDRL Compression Performance*

Table 8 shows the compression performance of Group 4 and EDRL on the CCITT test documents used to select the Group 4 algorithm. Each document represents a single page scanned at 400 dpi. Group 4's superior performance is due to its entropy-coded run-lengths, tuned to 400 dpi features.

Table 8. Group 4 and EDRL compression performance on standard CCITTT documents at 400 dpi

CCITT document number	Group 4 compression ratio	EDRL compression ratio
1	29.1	21.6
2	49.9	41.3

Table 8. Group 4 and EDRL compression performance on standard CCITTT documents at 400 dpi

CCITT document number	Group 4 compression ratio	EDRL compression ratio
3	17.9	14.1
4	7.3	5.5
5	15.8	12.4
6	31.0	25.5
7	7.4	5.3
8	26.7	23.4

Magazine text is typically typeset in a typeface with serifs (such as Times) at a point size of 10. At this size an A4/Letter page holds up to 14,000 characters, though a typical magazine page holds only about 7,000 characters. Text is seldom typeset at a point size smaller than 5. At 800 dpi, text cannot be meaningfully rendered at a point size lower than 2 using a standard typeface. Table 9 illustrates the legibility of various point sizes.

Table 9. Text at different point sizes

point size	sample text (in Times)
2	
3	
4	
5	The quick brown fox jumps over the lazy dog.
6	The quick brown fox jumps over the lazy dog.
7	The quick brown fox jumps over the lazy dog.
8	The quick brown fox jumps over the lazy dog.
9	The quick brown fox jumps over the lazy dog.
10	The quick brown fox jumps over the lazy dog.

Table 10 shows Group 4 and EDRL compression performance on pages of text of varying point sizes, rendered at 800 dpi. Note that EDRL achives the required compression ratio of 2.5 for an *entire page* of text typeset at a point size of 3. The distribution of characters on the test pages is based on English-language statistics [18].

Table 10. Group 4 and EDRL compression performance on text at 800 dpi

point size	characters/ A4 page	Group 4 compression ratio	EDRL compression ratio
2	340,000	2.3	1.7
3	170,000	3.2	2.5
4	86,000	4.7	3.8
5	59,000	5.5	4.9
6	41,000	6.5	6.1
7	28,000	7.7	7.4
8	21,000	9.1	9.0
9	17,000	10.2	10.4
10	14,000	10.9	23.2
11	12,000	11.5	12.4

Table 10. Group 4 and EDRL compression performance on text at 800 dpi

point size	characters/ A4 page	Group 4 compression ratio	EDRL compression ratio
12	8,900	13.5	14.8
13	8,200	13.5	15.0
14	7,000	14.6	16.6
15	5,800	16.1	18.5
20	3,400	19.8	23.9

For a point size of 9 or greater, EDRL slightly outperforms Group 4, simply because Group 4's runlength codes are tuned to 400 dpi.

These compression results bear out the observation that entropy-encoded runlengths contribute much less to compression than 2D encoding, unless the data is poorly correlated vertically, such as in the case of very small characters.

5.2.2 Contone Layer Compression

5.2.2.1 JPEG Compression

The JPEG compression algorithm [7] lossily compresses a contone image at a specified quality level. It introduces imperceptible image degradation at compression ratios below 5:1, and negligible image degradation at compression ratios below 10:1 [22].

JPEG typically first transforms the image into a color space which separates luminance and chrominance into separate color channels. This allows the chrominance channels to be subsampled without appreciable loss because of the human visual system's relatively greater sensitivity to luminance than chrominance. After this first step, each color channel is compressed separately.

The image is divided into into 8×8 pixel blocks. Each block is then transformed into the frequency domain via a discrete cosine transform (DCT). This transformation has the effect of concentrating image energy in relatively lower-frequency coefficients, which allows higher-frequency coefficients to be more crudely quantized. This quantization is the principal source of compression in JPEG. Further compression is achieved by ordering coefficients by frequency to maximise the likelihood of adjacent zero coefficients, and then runlength-encoding runs of zeroes. Finally, the runlengths and non-zero frequency coefficients are entropy coded. Decompression is the inverse process of compression.

5.2.2.2 CMYK Contone JPEG Compression Format

The CMYK contone layer is compressed to an interleaved color JPEG bytestream. The interleaving is required for space-efficient decompression in the printer, but may restrict the decoder to two sets of Huffman tables rather than four (i.e. one per color channel) [22]. If luminance and chrominance are separated, then the luminance channels can share one set of tables, and the chrominance channels the other set.

If luminance/chrominance separation is deemed necessary, either for the purposes of table sharing or for chrominance subsampling, then CMY is converted to YCrCb and Cr and Cb are duly subsampled. K is treated as a luminance channel and is not subsampled.

The JPEG bytestream is complete and self-contained. It contains all data required for decompression, including quantization and Huffman tables.

6 Memjet Printhead

An 8-inch Memjet printhead consists of two 4-inch printheads joined together, as shown in Figure 15. Each 4-inch printhead is responsible for printing half of the page.

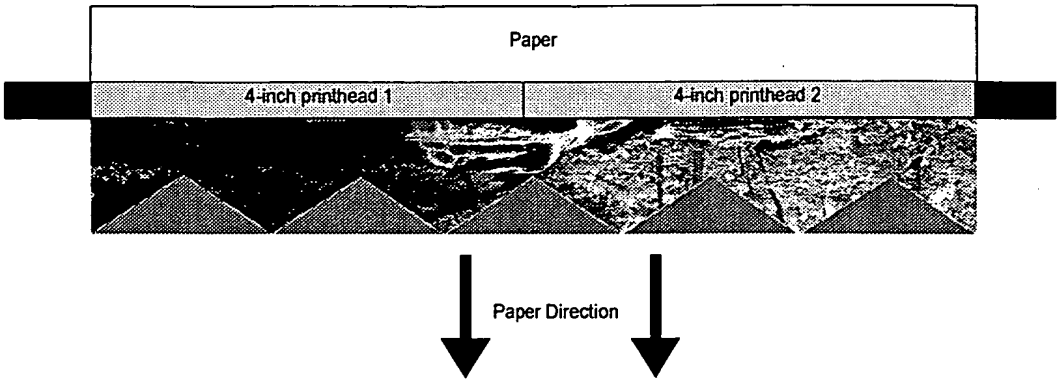


Figure 15. An 8-inch Memjet printhead is composed of two 4-inch Memjet printheads

The two 4-inch printheads are wired up together in a specific way for use in iPrint. Since the wiring requires knowledge of the 4-inch printhead, an overview of the 4-inch printhead is presented here.

6.1 Composition of a 4-inch Printhead

Each 4-inch printhead consists of 8 segments, each segment 1/2 an inch in length. Each of the segments prints bi-level cyan, magenta, yellow and black dots over a different part of the page to produce the final image. The positions of the segments are shown in Figure 16.

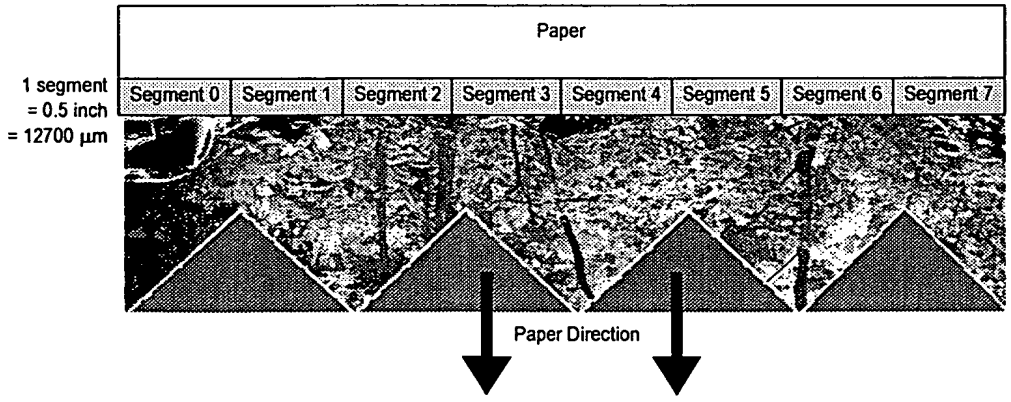


Figure 16. Arrangement of Segments in a 4-inch Printhead

Since the printhead prints dots at 1600 dpi, each dot is approximately 16μm in diameter. Thus each half-inch segment prints 800 dots, with the 8 segments corresponding to positions:.

Table 11. Final image dots addressed by each segment

Segment	Printhead 1		Printhead 2	
	First dot	Last dot	First dot	Last dot
0	0	799	6,400	7,199
1	800	1,599	7,200	7,999
2	1,600	2,399	8,000	8,799
3	2,400	3,199	8,800	9,599
4	3,200	3,999	9,600	10,399
5	4,000	4,799	10,400	11,199
6	4,800	5,599	11,200	11,999
7	5,600	6,399	12,000	12,799

Although each segment produces 800 dots of the final image, each dot is represented by a combination of bi-level cyan, magenta, yellow and black ink. Because the printing is bi-level, the input image should be dithered or error-diffused for best results.

Each segment then contains 3,200 nozzles: 800 each of cyan, magenta, yellow and black. A four-inch printhead contains 8 such segments for a total of 25,600 nozzles.

6.1.1 Grouping of Nozzles Within a Segment

The nozzles within a single segment are grouped for reasons of physical stability as well as minimization of power consumption during printing. In terms of physical stability, a total of 10 nozzles share the same ink reservoir. In terms of power consumption, groupings are made to enable a low-speed and a high-speed printing mode.

Memjet supports two printing speeds to allow speed/power consumption trade-offs to be made in different product configurations.

In the low-speed printing mode, 128 nozzles are fired simultaneously from each 4-inch printhead. The fired nozzles should be maximally distant, so 16 nozzles are fired from each segment. To fire all 25,600 nozzles, 200 different sets of 128 nozzles must be fired.

In the high-speed printing mode, 256 nozzles are fired simultaneously from each 4-inch printhead. The fired nozzles should be maximally distant, so 32 nozzles are fired from each segment. To fire all 25,600 nozzles, 100 different sets of 256 nozzles must be fired.

6.1.1.1 10 Nozzles Make a Pod

A single pod consists of 10 nozzles sharing a common ink reservoir. 5 nozzles are in one row, and 5 are in another. Each nozzle produces dots 16μm in diameter. Figure 17 shows the arrangement of a single pod, with the nozzles numbered according to the order in which they must be fired.

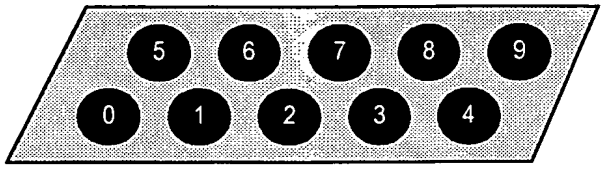


Figure 17. A single pod, numbered by firing order

Although the nozzles are fired in this order, the relationship of nozzles and physical placement of dots on the printed page is different. The nozzles from one row represent the even dots from one line on the page, and the nozzles on the other row represent the odd dots from the adjacent line on the page. Figure 18 shows the same pod with the nozzles numbered according to the order in which they must be loaded.

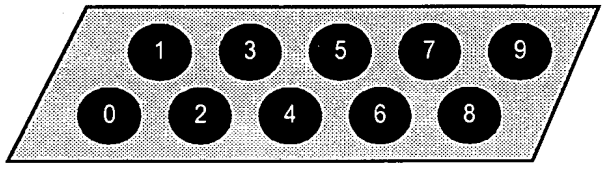


Figure 18. A single pod, numbered by logical order

The nozzles within a pod are therefore logically separated by the width of 1 dot. The exact distance between the nozzles will depend on the properties of the Memjet firing mechanism. In the best case, the printhead could be designed with staggered nozzles designed to match the flow of paper. In the worst case there is an error of 1/3200 dpi.

6.1.1.2 4 Pods Make a Quadpod

Four pods, one each of cyan, magenta, yellow and black, are grouped into a ***quadpod***. A quadpod represents the same horizontal set of 10 dots, but on different lines. The exact distance between different color pods depends on the Memjet operating parameters, and may vary from one Memjet design to another. The distance is considered to be a constant number of dot-widths, and must therefore be taken into account when printing: the dots printed by the cyan nozzles will be for different lines than those printed by the magenta,

yellow or black nozzles. The printing algorithm must allow for a variable distance up to about 8 dot-widths between colors. Figure 19 illustrates a single quadpod.

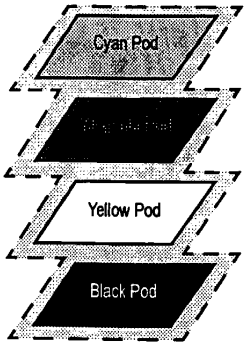


Figure 19. A Single Quadpod Contains 1 Pod of each Color

6.1.1.3 5 Quadpods Make a Podgroup

5 quadpods are organized into a single ***podgroup***. Since each quadpod contains 40 nozzles, each podgroup contains 200 nozzles: 50 cyan, 50 magenta, 50 yellow, and 50 black nozzles. The arrangement is shown in Figure 20, with quadpods numbered 0-4. Note that the distance between adjacent quadpods is exaggerated for clarity.

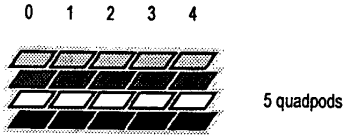


Figure 20. A Single Podgroup Contains 5 Quadpods

6.1.1.4 2 Podgroups Make a Bipodgroup

2 podgroups are organized into a single ***bipodgroup***. The formation of a bipodgroup is entirely for the purposes of low-speed and high-speed printing via 2 PodgroupEnable lines.

During low-speed printing, only one of the two PodgroupEnable lines is set in a given firing pulse, so only one podgroup of the two fires nozzles. During high-speed printing, both PodgroupEnable lines are set, so both podgroups fire nozzles. Consequently a low-speed print takes twice as long as a high-speed print, since the high-speed print fires twice as many nozzles at once.

Figure 21 illustrates the composition of a bipodgroup. The distance between adjacent podgroups is exaggerated for clarity.

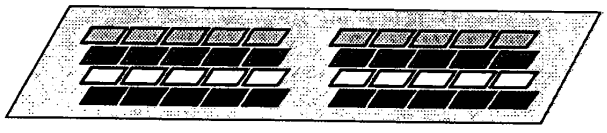


Figure 21. A single bipodgroup contains 2 podgroups

6.1.1.5 2 Bipodgroups Make a Firegroup

Two bipodgroups (BipodgroupA and BipodgroupB) are organized into a single ***firegroup***, with 4 firegroups in each segment. Each segment contains 4 firegroups. Firegroups are so named because they all fire the same nozzles simultaneously. Two enable lines, AEnable and BEnable, allow the firing of BipodgroupA nozzles and BipodgroupB nozzles independently. The arrangement is shown in Figure 22. The distance between adjacent firegroups is exaggerated for clarity.

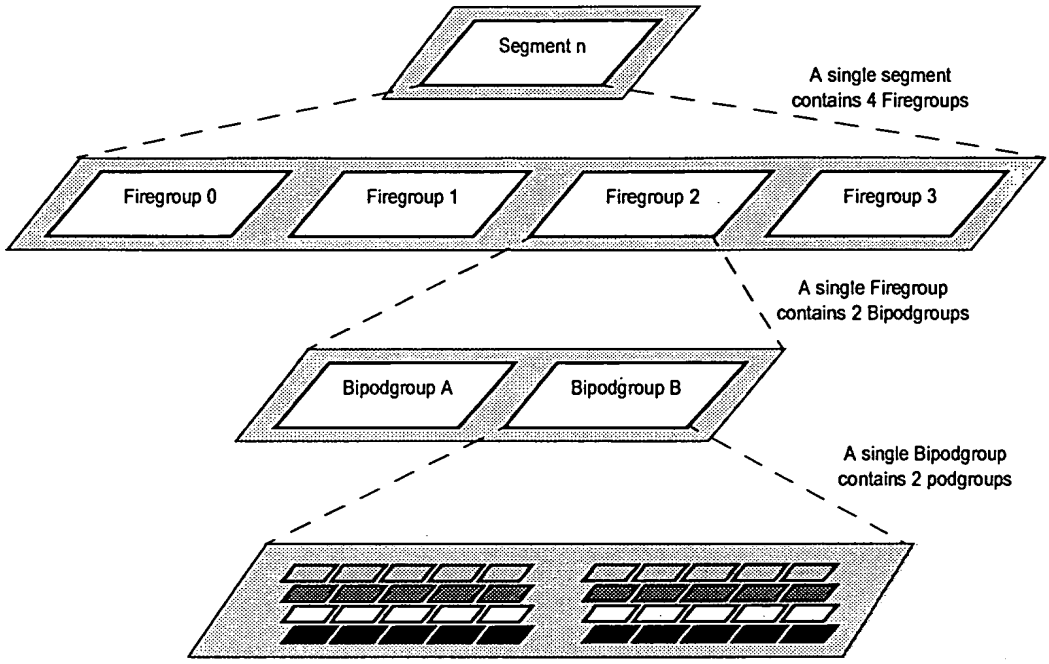


Figure 22. Relationship Between Segments, Firegroups, Bipodgroups, Podgroups and Quadpods

6.1.1.6 Nozzle Grouping Summary

Table 12 is a summary of the nozzle groupings in a printhead.

Table 12. Nozzle Groupings for a single 4-inch printhead

Name of Grouping	Composition	Replication Ratio	Nozzle Count
Nozzle	Base unit	1:1	1
Pod	Nozzles per pod	10:1	10
Quadpod	Pods per CMYK quadpod	4:1	40
Podgroup	Quadpods per podgroup	5:1	200
Bipodgroup	Podgroups per bipodgroup	2:1	400
Firegroup	Bipodgroups per firegroup	2:1	800
Segment	Firegroups per segment	4:1	3,200
4-inch printhead	Segments per 4-inch printhead	8:1	25,600

An 8-inch printhead consists of two 4-inch printheads for a total of 51,200 nozzles.

6.1.2 Load and Print Cycles

A single 4-inch printhead contains a total of 25,600 nozzles. A *Print Cycle* involves the firing of up to all of these nozzles, dependent on the information to be printed. A *Load Cycle* involves the loading up of the printhead with the information to be printed during the subsequent Print Cycle.

Each nozzle has an associated ***NozzleEnable*** bit that determines whether or not the nozzle will fire during the Print Cycle. The NozzleEnable bits (one per nozzle) are loaded via a set of shift registers.

Logically there are 4 shift registers per segment (one per color), each 800 deep. As bits are shifted into the shift register for a given color they are directed to the lower and upper nozzles on alternate pulses. Internally, each 800-deep shift register is comprised of two 400-deep shift registers: one for the upper nozzles, and one for the lower nozzles. Alternate bits are shifted into the alternate internal registers. As far as the external interface is concerned however, there is a single 800 deep shift register.

Once all the shift registers have been fully loaded (800 load pulses), all of the bits are transferred in parallel to the appropriate NozzleEnable bits. This equates to a single parallel transfer of 25,600 bits. Once the transfer has taken place, the Print Cycle can begin. The Print Cycle and the Load Cycle can occur simultaneously as long as the parallel load of all NozzleEnable bits occurs at the end of the Print Cycle.

6.1.2.1 Load Cycle

The Load Cycle is concerned with loading the printhead's shift registers with the next Print Cycle's NozzleEnable bits.

Each segment has 4 inputs directly related to the cyan, magenta, yellow and black shift registers. These inputs are called ***CDataIn***, ***MDataIn***, ***YDataIn*** and ***KDataIn***. Since there are 8 segments, there are a total of 32 color input lines per 4-inch printhead. A single pulse on the ***SRClock*** line (shared between all 8 segments) transfers the 32 bits into the appropriate shift registers. Alternate pulses transfer bits to the lower and upper nozzles respectively. Since there are 25,600 nozzles, a total of 800 pulses are required for the transfer. Once all 25,600 bits have been transferred, a single pulse on the shared ***PTransfer*** line causes the parallel transfer of data from the shift registers to the appropriate NozzleEnable bits.

The parallel transfer via a pulse on PTransfer must take place *after* the Print Cycle has finished. Otherwise the NozzleEnable bits for the line being printed will be incorrect.

Since all 8 segments are loaded with a single SRClock pulse, any printing process must produce the data in the correct sequence for the printhead. As an example, the first SRClock pulse will transfer the CMYK bits for the next Print Cycle's dot 0, 800, 1600, 2400, 3200, 4000, 4800, and 5600. The second SRClock pulse will transfer the CMYK bits for the next Print Cycle's dot 1, 801, 1601, 2401, 3201, 4001, 4801 and 5601. After 800 SRClock pulses, the PTransfer pulse can be given.

It is important to note that the odd and even CMYK outputs, although printed during the same Print Cycle, do not appear on the same physical output line. The physical separation of odd and even nozzles within the printhead, as well as separation between nozzles of different colors ensures that they will produce dots on different lines of the page. This relative difference must be accounted for when loading the data into the printhead. The actual difference in lines depends on the characteristics of the inkjet used in the printhead. The

differences can be defined by variables D_1 and D_2 where D_1 is the distance between nozzles of different colors, and D_2 is the distance between nozzles of the same color. Table 13 shows the dots transferred to segment *n* of a printhead on the first 4 pulses.

Table 13. Order of Dots Transferred to a 4-inch Printhead

Pulse	Dot	Black Line	Yellow Line	Magenta Line	Cyan Line
1	800S[a]	N	N+D_1[b]	N+2D_1	N+3D_1
2	800S+1	N+D_2[c]	N+D_1+D_2	N+2D_1+D_2	N+3D_1+D_2
3	800S+2	N	N+D_1	N+2D_1	N+3D_1
4	800S+3	N+D_2	N+D_1+D_2	N+2D_1+D_2	N+3D_1+D_2

a. S = segment number (0-7)

b. D_1 = number of lines between the nozzles of one color and the next (likely = 4-8)

c. D_2 = number of lines between two rows of nozzles of the same color (likely = 1)

And so on for all 800 pulses.

Data can be clocked into the printhead at a maximum rate of 20 MHz, which will load the entire data for the next line in 40µs.

6.1.2.2 Print Cycle

A 4-inch printhead contains 25,600 nozzles. To fire them all at once would consume too much power and be problematic in terms of ink refill and nozzle interference. Consequently two firing modes are defined: a low-speed printing mode and a high-speed printing mode:

- In the low-speed print mode, there are 200 phases, with each phase firing 128 nozzles. This equates to 16 nozzles per segment, or 4 per firegroup.
- In the high-speed print mode, there are 100 phases, with each phase firing 256 nozzles. This equates to 32 nozzles per segment, or 8 per firegroup.

The nozzles to be fired in a given firing pulse are determined by

- 3 bits ***QuadpodSelect*** (select 1 of 5 quadpods from a firegroup)
- 4 bits ***NozzleSelect*** (select 1 of 10 nozzles from a pod)
- 2 bits of ***PodgroupEnable*** lines (select 0, 1, or 2 podgroups to fire)

When one of the PodgroupEnable lines is set, only the specified Podgroup's 4 nozzles will fire as determined by QuadpodSelect and NozzleSelect. When both of the PodgroupEnable lines are set, both of the podgroups will fire their nozzles. For the low-speed mode, two fire pulses are required, with PodgroupEnable = 10 and 01 respectively. For the high-speed mode, only one fire pulse is required, with PodgroupEnable = 11.

The duration of the firing pulse is given by the ***AEnable*** and ***BEnable*** lines, which fire the PodgroupA and PodgroupB nozzles from all firegroups respectively. The typical duration of a firing pulse is 1.3 - 1.8 µs. The duration of a pulse depends on the viscosity of the ink (dependent on temperature and ink characteristics) and the amount of power available to the printhead. See Section 6.1.3 on page 33 for details on feedback from the printhead in order to compensate for temperature change.

The AEnable and BEnable are separate lines in order that the firing pulses can overlap. Thus the 200 phases of a low-speed Print Cycle consist of 100 A phases and 100 B phases, effectively giving 100 sets of Phase A and Phase B. Likewise, the 100 phases of a high-speed print cycle consist of 50 A phases and 50 B phases, effectively giving 50 phases of phase A and phase B.

Figure 23 shows the AEnable and BEnable lines during a typical Print Cycle. In a high-speed print there are 50 2µs cycles, while in a low-speed print there are 100 2µs cycles.

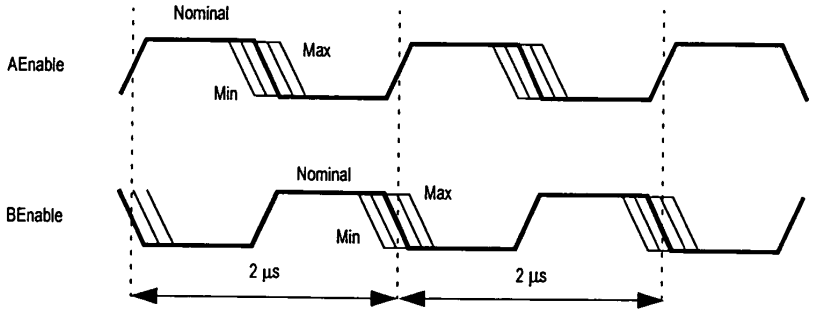


Figure 23. AEnable and BEnable During a Typical Print Cycle

For the high-speed printing mode, the firing order is:

- QuadpodSelect 0, NozzleSelect 0, PodgroupEnable 11 (Phases A and B)
- QuadpodSelect 1, NozzleSelect 0, PodgroupEnable 11 (Phases A and B)
- QuadpodSelect 2, NozzleSelect 0, PodgroupEnable 11 (Phases A and B)
- QuadpodSelect 3, NozzleSelect 0, PodgroupEnable 11 (Phases A and B)
- QuadpodSelect 4, NozzleSelect 0, PodgroupEnable 11 (Phases A and B)
- QuadpodSelect 0, NozzleSelect 1, PodgroupEnable 11 (Phases A and B)
- ...
- QuadpodSelect 3, NozzleSelect 9, PodgroupEnable 11 (Phases A and B)
- QuadpodSelect 4, NozzleSelect 9, PodgroupEnable 11 (Phases A and B)

For the low-speed printing mode, the firing order is similar. For each phase of the high speed mode where PodgroupEnable was 11, two phases of PodgroupEnable = 01 and 10 are substituted as follows:

- QuadpodSelect 0, NozzleSelect 0, PodgroupEnable 01 (Phases A and B)
- QuadpodSelect 0, NozzleSelect 0, PodgroupEnable 10 (Phases A and B)
- QuadpodSelect 1, NozzleSelect 0, PodgroupEnable 01 (Phases A and B)
- QuadpodSelect 1, NozzleSelect 0, PodgroupEnable 10 (Phases A and B)
- ...
- QuadpodSelect 3, NozzleSelect 9, PodgroupEnable 01 (Phases A and B)
- QuadpodSelect 3, NozzleSelect 9, PodgroupEnable 10 (Phases A and B)
- QuadpodSelect 4, NozzleSelect 9, PodgroupEnable 01 (Phases A and B)
- QuadpodSelect 4, NozzleSelect 9, PodgroupEnable 10 (Phases A and B)

When a nozzle fires, it takes approximately 100 μs to refill. The nozzle cannot be fired before this refill time has elapsed. This limits the fastest printing speed to 100μs per line. In the high-speed print mode, the time to print a line is 100μs, so the time between firing a nozzle from one line to the next matches the refill time. The low-speed print mode is slower than this, so is also acceptable.

The firing of a nozzle also causes perturbations for a limited time within the common ink reservoir of that nozzle's pod. The perturbations can interfere with the firing of another nozzle within the same pod. Consequently, the firing of nozzles within a pod should be offset from each other as long as possible. We therefore fire four nozzles from a quadpod (one nozzle per color) and then move onto the next quadpod within the podgroup.

- In the low-speed printing mode the podgroups are fired separately. Thus the 5 quadpods within both podgroups must all fire before the first quadpod fires again, totalling $10 \times 2\mu s$ cycles. Consequently each pod is fired once per 20μs.
- In the high-speed printing mode, the podgroups are fired together. Thus the 5 quadpods within a single podgroups must all fire before the first quadpod fires again, totalling $5 \times 2\mu s$ cycles. Consequently each pod is fired once per 10 μs.

6.1.3 Feedback from the Printhead

The printhead produces several lines of feedback (accumulated from the 8 segments). The feedback lines are used to adjust the timing of the firing pulses.Although each segment produces the same feedback, the feedback from all segments share the same tri-state bus lines. Consequently only one segment at a time can provide feedback.

A pulse on the ***SenseEnable*** line ANDed with data on CYAN enables the sense lines for that segment. The feedback sense lines are as follows:

- ***Tsense*** informs the controller how hot the printhead is. This allows the controller to adjust timing of firing pulses, since temperature affects the viscosity of the ink.
- ***Vsense*** informs the controller how much voltage is available to the actuator. This allows the controller to compensate for a flat battery or high voltage source by adjusting the pulse width.
- ***Rsense*** informs the controller of the resistivity (Ohms per square) of the actuator heater. This allows the controller to adjust the pulse widths to maintain a constant energy irrespective of the heater resistivity.
- ***Wsense*** informs the controller of the width of the critical part of the heater, which may vary up to ± 5% due to lithographic and etching variations. This allows the controller to adjust the pulse width appropriately.

6.1.4 Preheat Cycle

The printing process has a strong tendency to stay at the equilibrium temperature. To ensure that the first section of the printed photograph has a consistent dot size, the equilibrium temperature must be met *before* printing any dots. This is accomplished via a preheat cycle.

The Preheat cycle involves a single Load Cycle to all nozzles with 1s (i.e. setting all nozzles to fire), and a number of short firing pulses to each nozzle. The duration of the pulse must be insufficient to fire the drops, but enough to heat up the ink. Altogether about 200

pulses for each nozzle are required, cycling through in the same sequence as a standard Print Cycle.

Feedback during the Preheat mode is provided by Tsense, and continues until equilibrium temperature is reached (about 30° C above ambient). The duration of the Preheat mode is around 50 milliseconds, and depends on the ink composition.

6.1.5 Cleaning Cycle

In order to reduce the chances of nozzles becoming clogged, a cleaning cycle should be undertaken before each print. Each nozzle must be fired a number of times into an absorbent sponge.

The cleaning cycle involves a single Load Cycle to all nozzles with 1s (i.e. setting all nozzles to fire), and a number of firing pulses to each nozzle. The nozzles are cleaned via the same nozzle firing sequence as a standard Print Cycle. The number of times that each nozzle is fired depends upon the ink composition.

6.1.6 Printhead Interface Summary

A single 4-inch printhead has the following connections:

Table 14. Four-inch Printhead Connections

Name	#Pins	Description
Quadpod Select	3	Select which quadpod will fire (0-4)
NozzleSelect	4	Select which nozzle from the pod will fire (0-9)
PodgroupEnable	2	Enable the podgroups to fire (choice of: 01, 10, 11)
AEnable	1	Firing pulse for podgroup A
BEnable	1	Firing pulse for podgroup B
CDataIn[0-7]	8	Cyan input to cyan shift register of segments 0-7
MDataIn[0-7]	8	Magenta input to magenta shift register of segments 0-7
YDataIn[0-7]	8	Yellow input to yellow shift register of segments 0-7
KDataIn[0-7]	8	Black input to black shift register of segments 0-7
SRClock	1	A pulse on SRClock (ShiftRegisterClock) loads the current values from CDataIn[0-7], MDataIn[0-7], YDataIn[0-7] and KDataIn into the 32 shift registers.
PTransfer	1	Parallel transfer of data from the shift registers to the internal NozzleEnable bits (one per nozzle).
SenseEnable	1	A pulse on SenseEnable ANDed with data on CDataIn[n] enables the sense lines for segment n.
Tsense	1	Temperature sense
Vsense	1	Voltage sense
Rsense	1	Resistivity sense
Wsense	1	Width sense
Logic GND	1	Logic ground
Logic PWR	1	Logic power
V-	Bus bars	
V+		
TOTAL	52	

Internal to the 4-inch printhead, each segment has the following connections to the bond pads:

Table 15. Four Inch Printhead Internal Segment Connections

Name	#Pins	Description
Quadpod Select	3	Select which quadpod will fire (0-4)
NozzleSelect	4	Select which nozzle from the pod will fire (0-9)
PodgroupEnable	2	Enable the podgroups to fire (choice of: 01, 10, 11)
AEnable	1	Firing pulse for podgroup A
BEnable	1	Firing pulse for podgroup B
CDataIn	1	Cyan input to cyan shift register
MDataIn	1	Magenta input to magenta shift register
YDataIn	1	Yellow input to yellow shift register
KDataIn	1	Black input to black shift register
SRClock	1	A pulse on SRClock (ShiftRegisterClock) loads the current values from CDataIn, MDataIn, YDataIn and KDataIn into the 4 shift registers.
PTransfer	1	Parallel transfer of data from the shift registers to the internal NozzleEnable bits (one per nozzle).
SenseEnable	1	A pulse on SenseEnable ANDed with data on CDataIn enables the sense lines for this segment.
Tsense	1	Temperature sense
Vsense	1	Voltage sense
Rsense	1	Resistivity sense
Wsense	1	Width sense
Logic GND	1	Logic ground
Logic PWR	1	Logic power
V-	21	
V+	21	
TOTAL	66	(66 × 8 segments = 528 for all segments)

6.2 8-INCH PRINTHEAD CONSIDERATIONS

An 8-inch Memjet printhead is simply two 4-inch printheads physically placed together, as shown in Figure 15. The printheads are wired together and share many common connections in order that the number of pins from a controlling chip is reduced and that the two printheads can print simultaneously. A number of details must be considered because of this.

6.2.1 Connections

Since firing of nozzles from the two printheads occurs simultaneously, the QuadpodSelect, NozzleSelect, AEnable and BEnable lines are shared. For loading the printheads with data, the 32 lines of CDataIn, MDataIn, YDataIn and KDataIn are shared, and 2 different SRClock lines are used to determine which of the two printheads is to be loaded. A single PTransfer pulse is used to transfer the loaded data into the NozzleEnable bits for both printheads. Similarly, the Tsense, Vsense, Rsense, and Wsense lines are shared, with 2 SenseEnable lines to distinguish between the two printheads.

Therefore the two 4-inch printheads share all connections except SRClock and SenseEnable. These two connections are repeated, once for each printhead. The actual connections are shown here in Table 16:

Table 16. 8-inch Printhead Connections

Name	#Pins	Description
Quadpod Select	3	Select which quadpod will fire (0-4)
NozzleSelect	4	Select which nozzle from the pod will fire (0-9)
PodgroupEnable	2	Enable the podgroups to fire (choice of: 01, 10, 11)
AEnable	1	Firing pulse for podgroup A
BEnable	1	Firing pulse for podgroup B
CDataIn[0-7]	8	Cyan input to cyan shift register of segments 0-7
MDataIn[0-7]	8	Magenta input to magenta shift register of segments 0-7
YDataIn[0-7]	8	Yellow input to yellow shift register of segments 0-7
KDataIn[0-7]	8	Black input to black shift register of segments 0-7
SRClock1	1	A pulse on SRClock (ShiftRegisterClock) loads the current values from CDataIn[0-7], MDataIn[0-7], YDataIn[0-7] and KDataIn[0-7] into the 32 shift registers for 4-inch printhead 1.
SRClock2	1	A pulse on SRClock (ShiftRegisterClock) loads the current values from CDataIn[0-7], MDataIn[0-7] YDataIn[0-7] and KDataIn[0-7] into the 32 shift registers for 4-inch printhead 2.
PTransfer	1	Parallel transfer of data from the shift registers to the internal NozzleEnable bits (one per nozzle).
SenseEnable1	1	A pulse on SenseEnable ANDed with data on CDataIn[n] enables the sense lines for segment n in 4-inch printhead 1.
SenseEnable2	1	A pulse on SenseEnable ANDed with data on CDataIn[n] enables the sense lines for segment n in 4-inch printhead 2.
Tsense	1	Temperature sense
Vsense	1	Voltage sense
Rsense	1	Resistivity sense
Wsense	1	Width sense
Logic GND	1	Logic ground
Logic PWR	1	Logic power

Table 16. 8-inch Printhead Connections

Name	#Pins	Description
V-	Bus bars	
V+		
TOTAL	54	

6.2.2 Timing

The joining of two 4-inch printheads and wiring of appropriate connections enables an 8-inch wide image to be printed as fast as a 4-inch wide image. However, there is twice as much data to transfer to the 2 printheads before the next line can be printed. Depending on the desired speed for the output image to be printed, data must be generated and transferred at an appropriate speeds in order to keep up.

6.2.2.1 Example

As an example, consider the timing of printing an 8" × 12" page in 2 seconds. In order to print this page in 2 seconds, the 8-inch printhead must print 19,200 lines (12 × 1600). Rounding up to 20,000 lines in 2 seconds yields a line time of 100 µs. A single Print Cycle and a single Load Cycle must both finish within this time. In addition, a physical process external to the printhead must move the paper an appropriate amount.

From the printing point of view, the high-speed print mode allows a 4-inch printhead to print an entire line in 100 µs. Both 4-inch printheads must therefore be run in high-speed print mode to print simultaneously. Therefore 512 nozzles fire per firing pulse, thereby enabling the printing of an 8-inch line within the specified time.

The 800 SRClock pulses to both 4-inch printheads (each clock pulse transferring 32 bits) must also take place within the 100 µs line time. If both printheads are loaded simultaneously (64 data lines), the length of an SRClock pulse cannot exceed 100 µs/800 = 125ns, indicating that the printhead must be clocked at 8MHz. If the two printheads are loaded one at a time (32 shared data lines), the length of an SRClock pulse cannot exceed 100 µs/1600 = 62.5ns. The printhead must therefore be clocked at 16MHz. In both instances, the average time to calculate each bit value (for each of the 51,200 nozzles) must not exceed 100µs / 51,200 = 2ns. This requires a processor running at one of:

- 500 MHz generating 1 bit per cycle
- 250 MHz generating 2 bits per cycle
- 125 MHz generating 4 bits per cycle

7 Printer Controller

7.1 Printer Controller Architecture

The printer controller consists of the iPrint central processor (ICP) chip, a 64MBit RDRAM, and the system authentication chip.

The ICP contains a general-purpose processor and a set of purpose-specific functional units controlled by the processor via the processor bus, as shown in Figure 24. Only three functional units are non-standard - the EDRL Bi-level Page Expander, the Halftoner/Compositor, and the Printhead Interface which controls the Memjet printhead.

Software running on the processor coordinates the various functional units to receive, expand and print pages. This is described in the next section.

The various functional units of the ICP are described in subsequent sections.

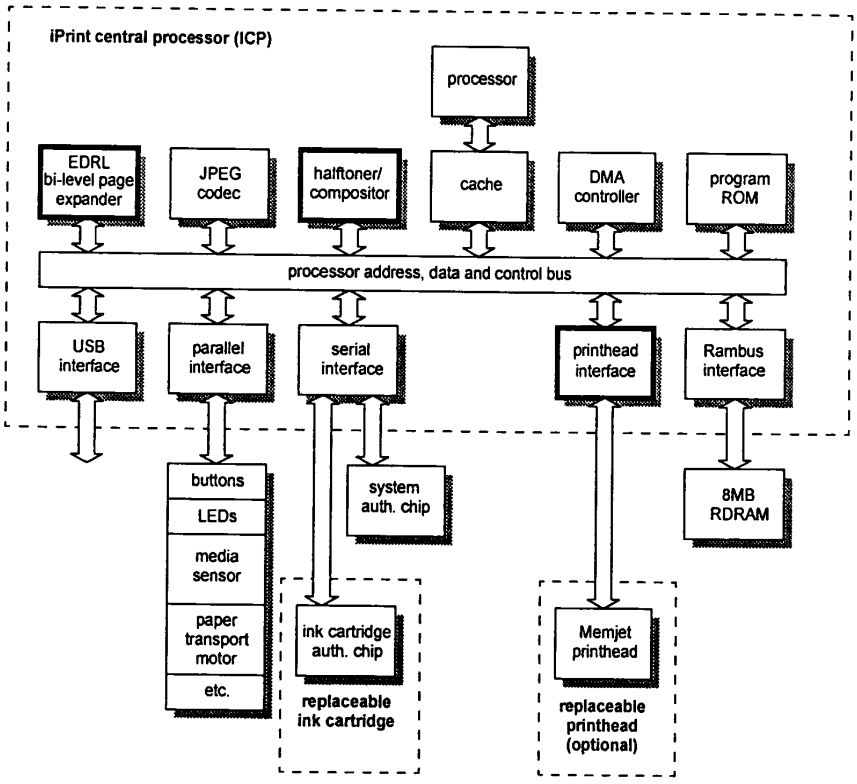


Figure 24. Printer controller architecture

7.2 Page Expansion and Printing

Page expansion and printing proceeds as follows. A page description is received from the host via the USB interface and is stored in main memory. 6MB of main memory is dedicated to page storage. This can hold two pages each not exceeding 3MB, or one page exceeding 3MB. If the host generates pages not exceeding 3MB, then the printer operates

in streaming mode - i.e. it prints one page while receiving the next. If the host generates pages exceeding 3MB, then the printer operates in single-page mode - i.e. it receives each page and prints it before receiving the next. If the host generates pages exceeding 6MB then they are rejected by the printer.

A page consists of two parts - the bi-level black layer, and the contone layer. These are compressed in distinct formats - the bi-level black layer in EDRL format, the contone layer in JPEG format. The first stage of page expansion consists of decompressing the two layers in parallel. The bi-level layer is decompressed by the EDRL bi-level page expander unit, the contone layer by the JPEG Codec.

The second stage of page expansion consists of halftoning the contone CMYK data to bi-level CMYK, and then compositing the bi-level black layer over the bi-level CMYK layer. The halftoning and compositing is carried out by the Halftoner/Compositor unit.

Finally, the composited bi-level CMYK image is printed via the printhead interface unit, which controls the Memjet printhead.

Because the Memjet printhead prints at high speed, the paper must move past the printhead at a constant velocity. If the paper is stopped because data can't be fed to the printhead fast enough, then visible printing irregularities will occur. It is therefore important to transfer bi-level CMYK data to the printhead interface at the required rate.

A fully-expanded 1600 dpi bi-level CMYK page has a size of 114.3MB. Because it is impractical to store an expanded page in printer memory, each page is expanded in real time during printing. Thus the various stages of page expansion and printing are pipelined. The page expansion and printing data flow is described in Table 17. The aggregate traffic to/from main memory of 174MB/s is well within the capabilities of current technologies such as Rambus.

Table 17. Page expansion and printing data flow

process	input	input window	output	output window	input rate	output rate
receive contone	-	-	JPEG stream	1	- -	1.5MB/s 3.3Mp/s
receive bi-level	-	-	EDRL stream	1	- -	1.5MB/s 30Mp/s
decompress contone	JPEG stream	-	32-bit CMYK	8	1.5MB/s 3.3Mp/s	13MB/s 3.3Mp/s
decompress bi-level	EDRL stream	-	1-bit K	1	1.5MB/s 30Mp/s[a]	14MB/s 120Mp/s
halftone	32-bit CMYK	1	⇓	⇓	13MB/s 3.3Mp/s[b]	⇓
composite	1-bit K	1	4-bit CMYK	1	14MB/s 120Mp/s	57MB/s 120Mp/s
print	4-bit CMYK	24, 1[c]	-	-	57MB/s 120Mp/s	- -
					87MB/s	87MB/s
						174MB/s

a. 800 dpi ⇒ 1600 dpi (2 × 2 expansion).
b. 267 ppi ⇒ 1600 dpi (6 × 6 expansion).
c. Needs a window of 24 lines, but only advances 1 line.

Each stage communicates with the next via a shared FIFO in main memory. Each FIFO is organised into lines, and the minimum size (in lines) of each FIFO is designed to accommodate the output window (in lines) of the producer and the input window (in lines) of the consumer. The inter-stage main memory buffers are described in Table 18. The aggregate buffer space usage of 6.3MB leaves 1.7MB free for program code and scratch memory (out of the 8MB available).

Table 18. Page expansion and printing main memory buffers

buffer	organisation and line size	number of lines	buffer size
compressed contone page	byte stream -	-	3MB
compressed bi-level page	byte stream -	-	3MB
contone CMYK buffer	32-bit interleaved CMYK (267 ppi x 8" x 32 = 8.3KB)	8 x 2 = 16	134KB
bi-level K buffer	1-bit K (800 dpi x 8" x 1 = 1.5KB)	1 x 2 = 2	3KB
bi-level CMYK buffer	4-bit planar odd/even CMYK (1600 dpi x 8" x 4 = 6.3KB)	24 + 1 = 25	156KB
			6.3MB

The overall data flow, including FIFOs, is illustrated in Figure 25.

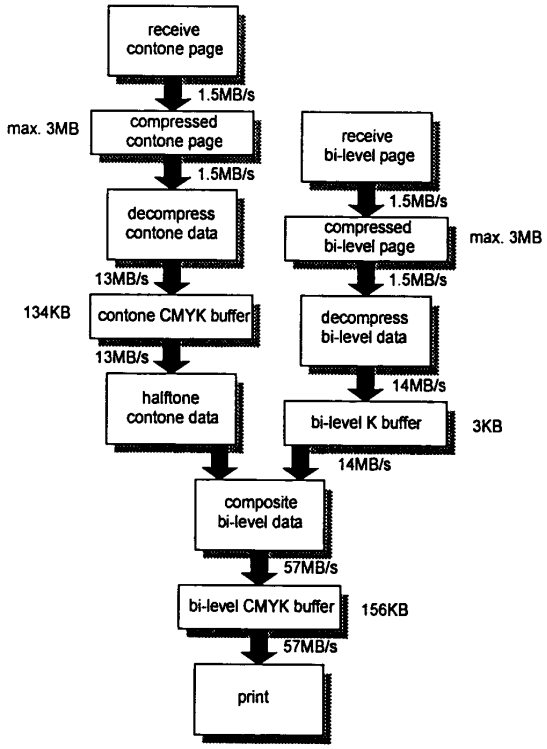


Figure 25. Page expansion and printing data flow summary

Contone page decompression is carried out by the JPEG Codec. Bi-level page decompression is carried out by the EDRL bi-level page expander. Halftonign and compositing is carried out by the Halftoner/Compositor unit. These functional units are described in the following sections.

7.2.1 DMA Approach

Each functional unit contains one or more on-chip input and/or output FIFOs. Each FIFO is allocated a separate channel in the multi-channel DMA controller. The DMA controller handles single-address rather than double-address transfers, and so provides a separate request/acknowledge interface for each channel.

Each functional unit stalls gracefully whenever an input FIFO is exhausted or an output FIFO is filled.

The processor programs each DMA transfer. The DMA controller generates the address for each word of the transfer on request from the functional unit connected to the channel. The functional unit latches the word onto or off the data bus when its request is acknowledged by the DMA controller. The DMA controller interrupts the processor when the transfer is complete, thus allowing the processor to program another transfer on the same channel in a timely fashion.

In general the processor will program another transfer on a channel as soon as the corresponding main memory FIFO is available (i.e. non-empty for a read, non-full for a write).

The granularity of channel servicing implemented in the DMA controller depends somewhat on the latency of main memory.

7.2.2 EDRL Bi-Level Page Expander

The EDRL Bi-Level Page Expander decompresses an EDRL-compressed bi-level image.

The input to the EDRL Expander is an EDRL bitstream. The output from the EDRL Expander is a set of bi-level image lines, scaled horizontally from the resolution of the expanded bi-level image by an integer scale factor to 1600 dpi.

Once started, the EDRL Expander proceeds until it detects an *end-of-page* code in the EDRL bitstream, or until it is explicitly stopped via its control register.

The EDRL Expander relies on an explicit page width to decode the bitstream. This must be written to the *page width* register prior to starting the EDRL Expander.

The scaling of the expanded bi-level image relies on an explicit scale factor. This must be written to the *scale factor* register prior to starting the EDRL Expander.

Table 19. EDRL Expander control and configuration registers

register	width	description
start	1	Start the EDRL Expander.
stop	1	Stop the EDRL Expander.
page width	13	Page width used during decoding to detect end-of-line.
scale factor	4	Scale factor used during scaling of expanded image.

The EDRL compression format is described in Section 5.2.1.2. It represents a bi-level image in terms of its edges. Each edge in each line is coded relative to an edge in the previous line, or relative to the previous edge in the same line. No matter how it is coded, each edge is ultimately decoded to its distance from the previous edge in the same line. This distance, or runlength, is then decoded to the string of one bits or zero bits which represent the corresponding part of the image. The decompression algorithm is defined in Section 5.2.1.2.

The EDRL Expander consists of a bitstream decoder, a state machine, edge calculation logic, two runlength decoders, and a runlength (re)encoder.

The bitstream decoder decodes an entropy-coded codeword from the bitstream and passes it to the state machine. The state machine returns the size of the codeword to the bitstream decoder, which allows the decoder to advance to the next codeword. In the case of a *create edge* code, the state machine uses the bitstream decoder to extract the corresponding runlength from the bitstream. The state machine controls the edge calculation logic and runlength decoding/encoding as defined in Table 21.

The edge calculation logic is quite simple. The current edge offset in the previous (reference) and current (coding) lines are maintained in the reference edge latch and edge latch respectively. The runlength associated with a *create edge* code is output directly to the run decoder, and is added to the current edge. A *delta* code is translated into a runlength by adding the associated delta to the reference edge and subtracting the current edge. The generated runlength is output to the run decoder, and is added to the current edge. The next runlength is extracted from the run encoder and added to the reference edge. A *kill edge* code simply causes the current reference edge to be skipped. Again the next runlength is extracted from the run encoder and added to the reference edge.

Each time the edge calculation logic generates a runlength representing an edge, it is passed to the run decoder. While the run decoder decodes the run it generates a stall signal to the state machine. Since the run decoder is slower than the edge calculation logic, there's not much point in decoupling it. The expanded line accumulates in a line buffer large enough to hold an 8" 800 dpi line (800 bytes).

The previously expanded line is also buffered. It acts as a reference for the decoding of the current line. The previous line is re-encoded as runlengths on demand. This is less expensive than buffering the decoded runlengths of the previous line, since the worst case is one 13-bit runlength for each pixel (20KB at 1600 dpi). While the run encoder encodes the run it generates a stall signal to the state machine. The run encoder uses the page width to detect end-of-line.

A second run decoder decodes the output runlength to a line buffer large enough to hold an 8" 1600 dpi line (1600 bytes). The runlength passed to this output run decoder is multiplied by the scale factor, so this decoder produces 1600 dpi lines. The line is output *scale factor* times through the output pixel FIFO. This achieves the required vertical scaling by simple line replication. The EDRL Expander could be designed with *edge smoothing* integrated into its image scaling. A simple smoothing scheme based on template-matching can be very effective [12]. This would require a multi-line buffer between the low-resolution run decoder and the smooth scaling unit, but would eliminate the high-resolution run decoder.

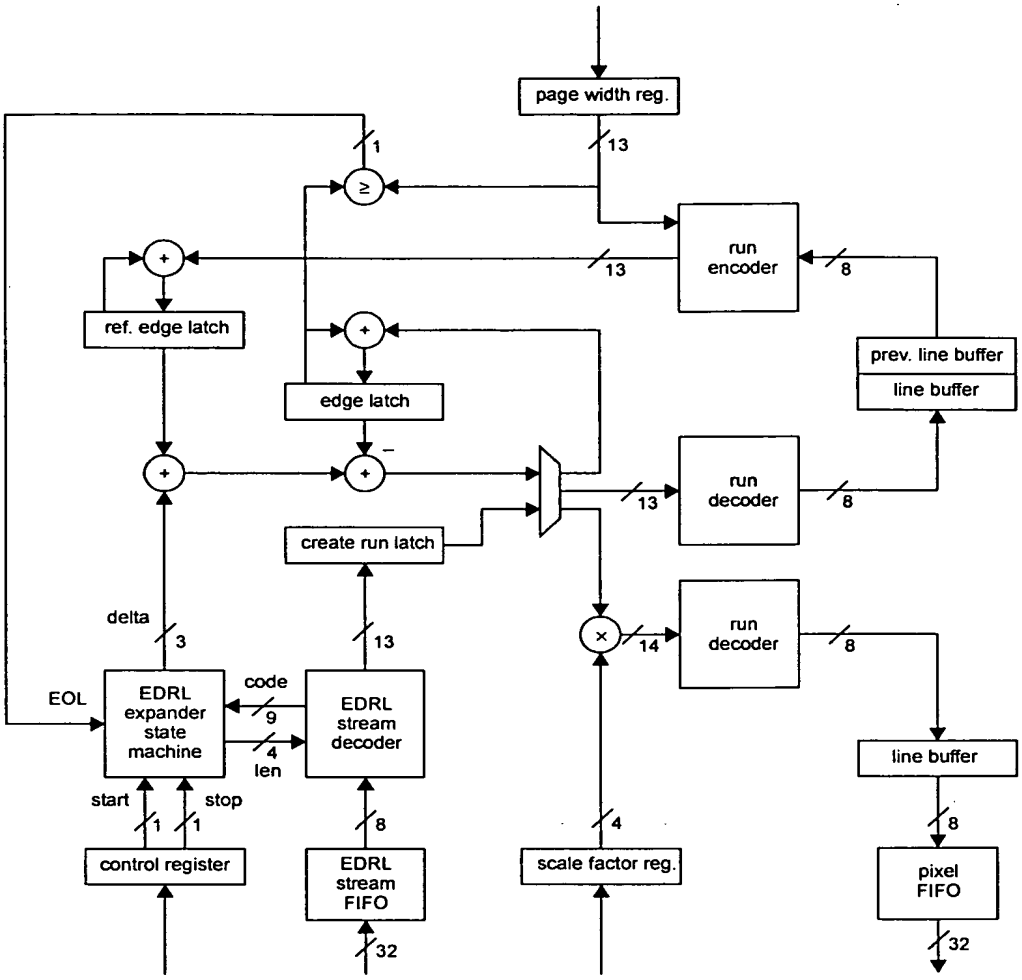


Figure 26. EDRL bi-level page expander

7.2.2.1 *EDRL Stream Decoder*

The EDRL Stream Decoder decodes entropy-coded EDRL codewords in the input bitstream. It uses a 16-bit barrel shifter whose left (most significant) edge is always aligned to a codeword boundary in the bitstream. The decoder connected to the barrel shifter decodes a codeword according to Table 20, and supplies the state machine with the corresponding code.

Table 20. EDRL stream codeword decoding table

input codeword bit pattern[a]	output code	output code bit pattern
1xxx xxxx	Δ0	1 0000 0000
010x xxxx	Δ+1	0 1000 0000
011x xxxx	Δ−1	0 0100 0000
0010 xxxx	kill edge	0 0010 0000

Table 20. EDRL stream codeword decoding table

Input codeword bit pattern[a]	output code	output code bit pattern
0011 xxxx	create near edge	0 0001 0000
0001 0xxx	Δ+2	0 0000 1000
0001 1xxx	Δ−2	0 0000 0100
0000 1xxx	create far edge	0 0000 0010
0000 01xx	end-of-page (EOP)	0 0000 0001

a. x = don't care

The state machine in turn outputs the length of the code. This is added, modulo-8, to the current codeword bit offset to yield the next codeword bit offset. The bit offset in turn controls the barrel shifter. If the codeword bit offset wraps, then the carry bit controls the latching of the next byte from the input FIFO. At this time bytes 1 and 2 are latched into the barrel shifter, byte 2 is latched to byte 1, and the FIFO output is latched to byte 2. The two byte latches are always one byte ahead of the barrel shifter, in readiness for the next bit offset wrap. It takes three cycles of length 8 to prime the barrel shifter. This is handled by starting states in the state machine.

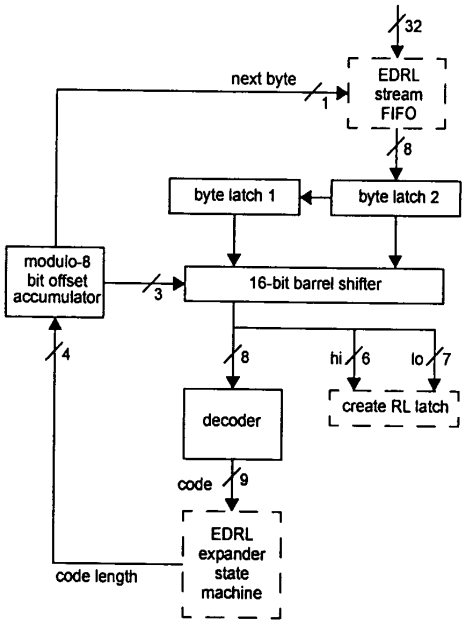


Figure 27. EDRL Stream Decoder

7.2.2.2 *EDRL Expander State Machine*

The EDRL Expander State Machine controls the edge calculation and runlength expansion logic in response to codes supplied by the EDRL Stream Decoder. It supplies the EDRL Stream Decoder with the length of the current codeword and supplies the edge calculation logic with the delta value associated with the current delta code. The state machine also responds to *start* and *stop* control signals from the control register, and the *end-of-line* (EOL) signal from the edge calculation logic.

The state machine also controls the multi-cycle fetch of the runlength associated with a *create edge* code.

Table 21. EDRL Expander State Machine

input signal	input code	current state	next state	code len	delta	actions
start	-	stopped	starting 1	8	-	-
-	-	starting 1	starting 2	8	-	-
-	-	starting 2	idle	8	-	-
stop	-	-	stopped	0	-	reset run decoders and FIFOs
EOL	-	-	EOL 1	0	-	swap line buffers; reset run decoders; reset ref. edge and edge
-	-	EOL 1	idle			run encoder ⇒ ref.run; ref.edge += ref.run;
-	Δ0	idle	idle	1	0	run = edge - ref.edge + delta; edge += run; run ⇒ run decoder; run encoder ⇒ ref.run; ref.edge += ref.run;
-	Δ+1	idle	idle	2	+1	"
-	Δ−1	idle	idle	3	−1	"
-	Δ+2	idle	idle	4	+2	"
-	Δ−2	idle	idle	5	−2	"
-	kill edge	idle	idle	6	-	run encoder ⇒ ref.run; ref.edge += ref.run;
-	create near edge	idle	create run lo 7	7	-	reset create run
-	create far edge	idle	create run hi 6	8	-	-
-	EOP	idle	stopped	8	-	-
-	-	create run hi 6	create run lo 7	6	-	latch cr. run hi 6
-	-	create run lo 7	create edge	7	-	latch cr. run lo 7
-	-	create edge	idle	0	-	run = create run; edge += run; run ⇒ run encoder;

7.2.3 JPEG Codec

The JPEG Codec decompresses a JPEG-compressed CMYK contone image.

The input to the JPEG Codec is a JPEG bitstream. The output from the JPEG Codec is a set of contone CMYK image lines.

When decompressing, the JPEG Codec writes its output in the form of 8×8 pixel blocks. These are sometimes converted to full-width lines via an *page width* × 8 strip buffer

closely coupled with the codec. This would require a 67KB buffer. We instead use 8 parallel pixel FIFOs with shared bus access and 8 corresponding DMA channels, as shown in Figure 28.

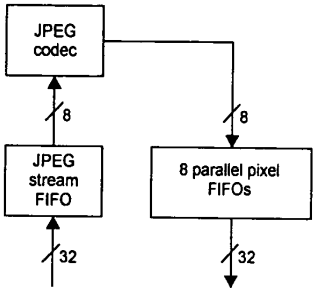


Figure 28. JPEG Codec

7.2.4 Halftoner/Compositor

The Halftoner/Compositor combines the functions of halftoning the contone CMYK layer to bi-level CMYK, and compositing the black layer over the halftoned contone layer.

The input to the Halftoner/Compositor is an expanded 267 ppi CMYK contone layer, and an expanded 1600 dpi black layer. The output from the Halftoner/Compositor is a set of 1600 dpi bi-level CMYK image lines.

Once started, the Halftoner/Compositor proceeds until it detects an *end-of-page* condition, or until it is explicitly stopped via its control register.

The Halftoner/Compositor relies on an explicit page width and page length to detect end-of-line and end-of-page respectively. These must be written to the *page width* and *page length* registers prior to starting the Halftoner/Compositor.

The clipping of lines to the *requested* page width (as opposed to the *physical* page width) relies on an explicit clip width. This must be written to the *clip width* register prior to starting the Halftoner/Compositor.

The scaling of the expanded contone image relies on an explicit scale factor. This must be written to the *contone scale factor* register prior to starting the Halftoner/Compositor.

Table 22. Halftoner/Compositor control and configuration registers

register	width	description
start	1	Start the Halftoner/Compositor.
stop	1	Stop the Halftoner/Compositor.
page width	14	Page width used to detect end-of-line.
page length	15	Page length used to detect end-of-page.
clip width	14	Clip width used to clip lines.
contone scale factor	4	Scale factor used to scale contone data to bi-level resolution.

The consumer of the data produced by the Halftoner/Compositor is the Printhead Interface. The Printhead Interface requires bi-level CMYK image data in *planar* format, i.e.

with the color planes separated. Further, it also requires that even and odd pixels are separated. The output stage of the Halftoner/Compositor therefore uses 8 parallel pixel FIFOs, one each for *even cyan*, *odd cyan*, *even magenta*, *odd magenta*, *even yellow*, *odd yellow*, *even black*, and *odd black*.

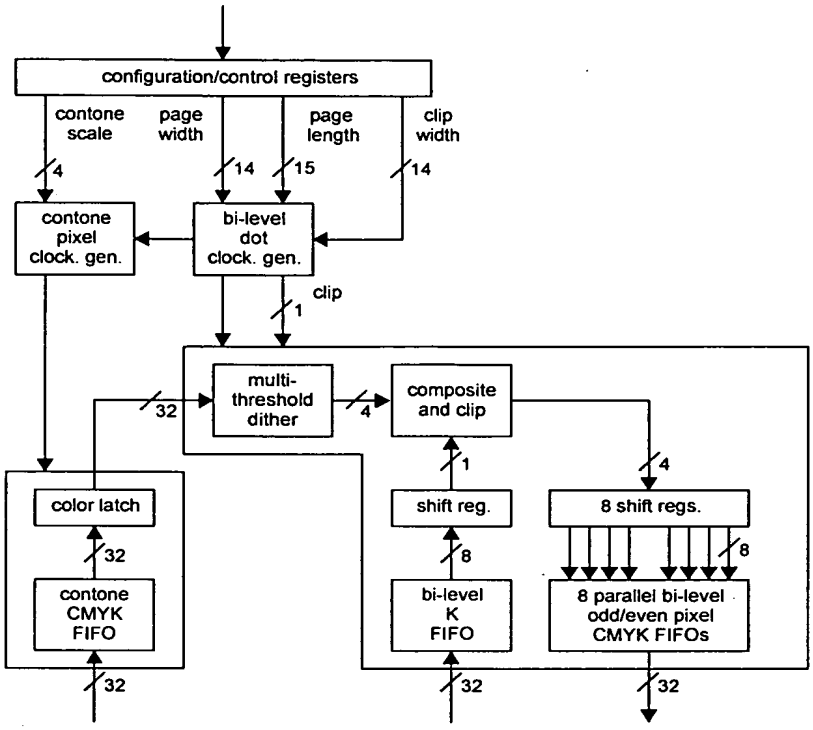


Figure 29. Halftoner/Compositor

The input contone CMYK FIFO is actually a full 8KB line buffer. The line is used once during loading in FIFO mode, and is then used another *contone scale factor - 1* times to effect vertical up-scaling via line replication. An alternative is to read the line from main memory *contone scale factor* times, increasing memory traffic by 65MB/s.

A general 256-layer *dither volume* provides great flexibility in dither cell design, by decoupling different intensity levels. General dither volumes can be large - a 64×64×256 dither volume, for example, has a size of 128KB. They are also inefficient to access since each color component requires the retrieval of a different bit from the volume. In practice, there is no need to fully decouple each layer of the dither volume. Each dot column of the volume can be implemented as a fixed set of thresholds rather than 256 separate bits. Using four 8-bit thresholds, for example, only consumes 32 bits. Now, n thresholds define $n+1$ intensity intervals, within which the corresponding dither cell location is alternately not set or set. The contone pixel value being dithered uniquely selects one of the $n+1$ intervals, and this determines the value of the corresponding output dot.

We dither the contone data using a four-threshold 16KB 64×64×32 dither volume. The four thresholds form a convenient 32-bit value which can be retrieved from the dither cell ROM in one cycle. If dither cell registration is desired between color planes, then the same 32-bit four-threshold value can be retrieved once and used to dither each color component. If dither cell registration is not desired, then the dither cell can be split into four subcells and stored in four separately addressable ROMs from which four different 32-bit four-

threshold values can be retrieved in parallel in one cycle. Using the addressing scheme shown below, the four color planes share the same dither cell at vertical and/or horizontal offsets of 32 dots from each other.

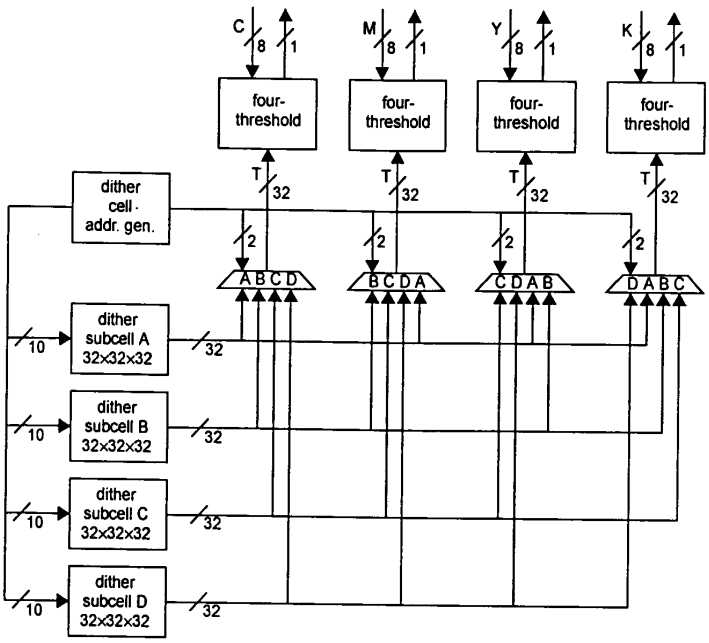


Figure 30. Four-threshold dither

The four-threshold unit converts a four-threshold value and an intensity value into an interval and thence a one or zero bit. The four-thresholding rules are shown in Table 23.

Table 23. Four-thresholding rules

interval	output
$V < T_1$	0
$T_1 < V \leq T_2$	1
$T_2 < V \leq T_3$	0
$T_3 < V \leq T_4$	1
$T_4 < V$	0

The corresponding logic is shown in Figure 31.

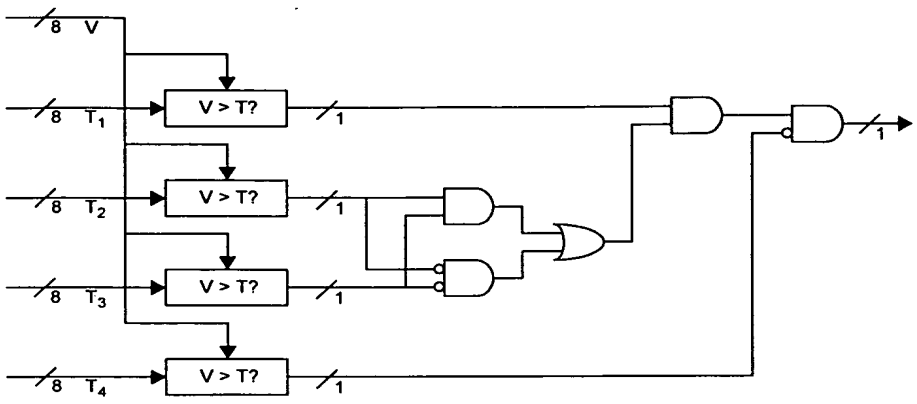


Figure 31. Four-threshold

The composite unit composites a black layer dot over a halftoned CMYK layer dot. If the black layer opacity is one, then the halftoned CMY is set to zero. If the *clip* signal is one, then the entire color output is zero.

Given a 4-bit halftoned color $C_cM_cY_cK_c$, a 1-bit black layer opacity K_b, and a 1-bit clip signal, the composite and clip logic is as defined in Table 24.

Table 24. Composite and clip logic

color channel	expression
C	$C_c \wedge \neg(clip \vee K_b)$
M	$M_c \wedge \neg(clip \vee K_b)$
Y	$Y_c \wedge \neg(clip \vee K_b)$
K	$(K_c \vee K_b) \wedge \neg clip$

7.3 PRINTHEAD INTERFACE

The Printhead Interface (PHI) is the means by which the Processor loads the Memjet printhead with the dots to be printed, and controls the actual dot printing process. The PHI contains:

- a Line Loader/Format Unit (LLFU) which loads the dots for a given print line into local buffer storage and formats them into the order required for the Memjet printhead.
- a Memjet Interface (MJI), which transfers data to the Memjet printhead, and controls the nozzle firing sequences during a print.

The units within the PHI are controlled by a number of registers that are programmed by the Processor. In addition, the Processor is responsible for setting up the appropriate parameters in the DMA Controller for the transfers from memory to the LLFU.

The internal structure of the Printhead Interface is shown in Figure 32.

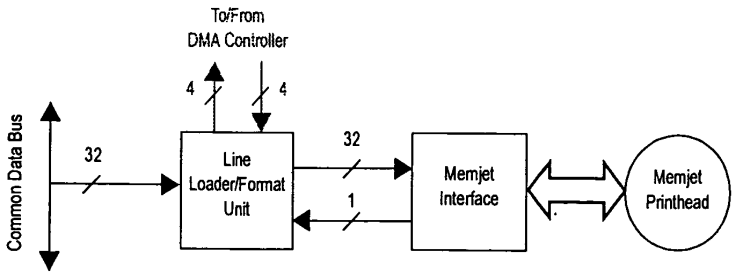


Figure 32. Internal Structure of Printhead Interface

7.3.1 Line Loader/Format Unit

The Line Loader/Format Unit (LLFU) loads the dots for a given print line into local buffer storage and formats them into the order required for the Memjet printhead. It is responsible for supplying the pre-calculated nozzleEnable bits to the Memjet Interface for the eventual printing of the page.

A single line in the 8-inch printhead consists of 12,800 4-color dots. At 1 bit per color, a single print line consists of 51,200 bits. These bits must be supplied in the correct order for being sent on to the printhead. See Section 6.1.2.1 on page 30 for more information concerning the Load Cycle dot loading order, but in summary, 32 bits are transferred at a time to each of the two 4-inch printheads, with the 32 bits representing 4 dots for each of the 8 segments.

The printing uses a double buffering scheme for preparing and accessing the dot-bit information. While one line is being loaded into the first buffer, the pre-loaded line in the second buffer is being read in Memjet dot order. Once the entire line has been transferred from the second buffer to the printhead via the Memjet interface, the reading and writing processes swap buffers. The first buffer is now read and the second buffer is loaded up with the new line of data. This is repeated throughout the printing process, as can be seen in the conceptual overview of Figure 33.

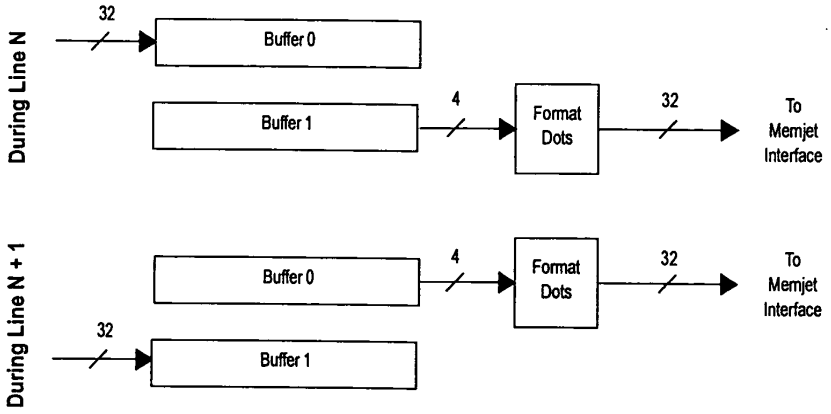


Figure 33. Conceptual Overview of Double Buffering During Print Lines N and N+1

The actual implementation of the LLFU is shown in Figure 34. Since one buffer is being read from while the other is being written to, two sets of address lines must be used. The 32-bits DataIn from the common data bus are loaded depending on the WriteEnables, which are generated by the State Machine in response to the DMA Acknowledges.

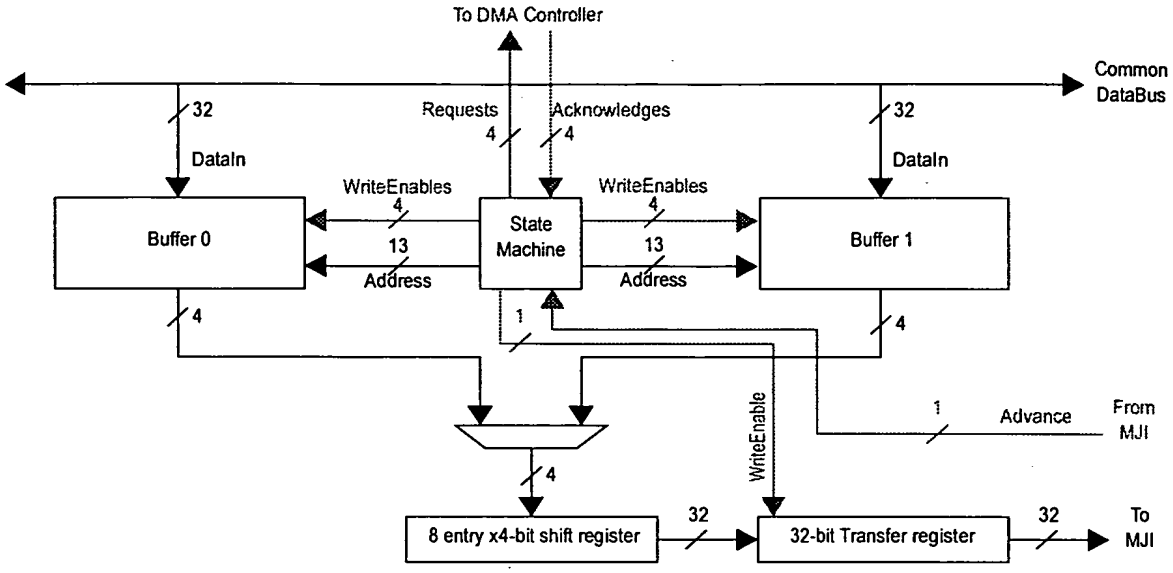


Figure 34. Structure of LLFU

A multiplexor chooses between the two 4-bit outputs of Buffer 0 and Buffer 1, and sends the result to an 8-entry by 4-bit shift register. After the first 8 read cycles, and whenever an Advance pulse comes from the MJI, the current 32-bit value from the shift register is gated into the 32-bit Transfer register, where it can be used by the MJI.

7.3.1.1 Buffers

Each of the two buffers is broken into 4 sub-buffers, 1 per color. All the even dots are placed before the odd dots in each color's buffer, as shown in Figure 35.

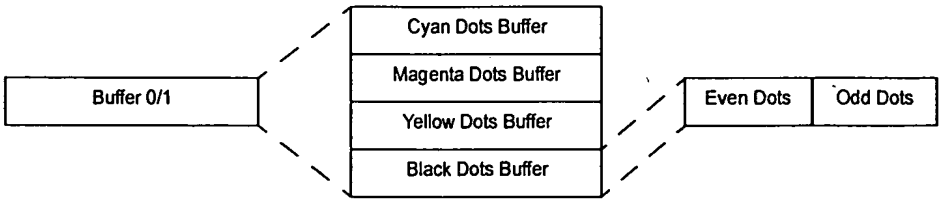


Figure 35. Conceptual Structure of Buffer

The 51,200 bits representing the dots in the next line to be printed are stored 12,800 bits per color buffer, stored as 400 32-bit words. The first 200 32-bit words (6400 bits) represent the even dots for the color, while the second 200 32-bit words (6400 bits) represent the odd dots for the color.

The addressing decoding circuitry is such that in a given cycle, a single 32-bit access can be made to all 4 sub-buffers - either a read from all 4 or a write to one of the 4. Only one

bit of the 32-bits read from each color buffer is selected, for a total of 4 output bits. The process is shown in Figure 36. 13 bits of address allow the reading of a particular bit by means of 8-bits of address being used to select 32 bits, and 5-bits of address choose 1-bit from those 32. Since all color buffers share this logic, a single 13-bit address gives a total of 4 bits out, one per color. Each buffer has its own WriteEnable line, to allow a single 32-bit value to be written to a particular color buffer in a given cycle. The 32-bits of DataIn are shared, since only one buffer will actually clock the data in.

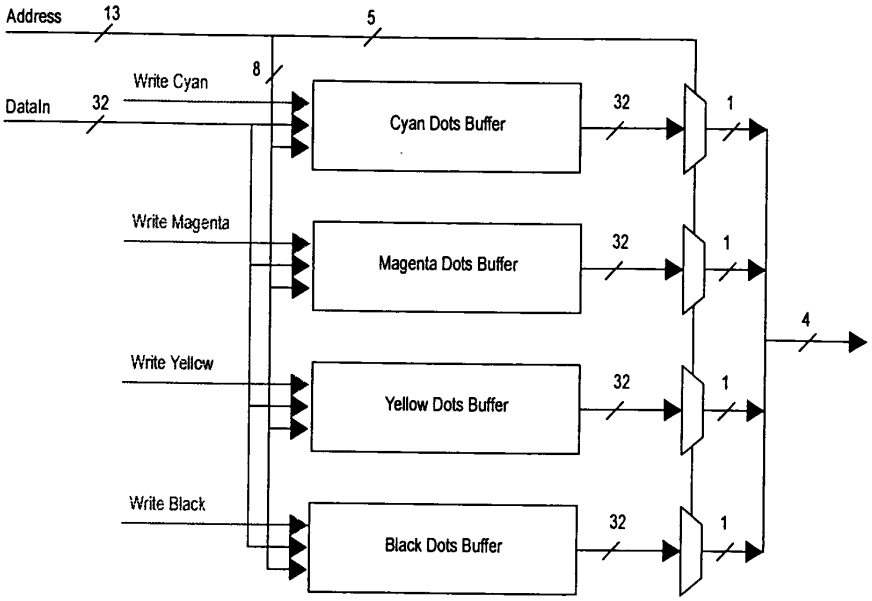


Figure 36. Logical Structure of Buffer

7.3.1.2 Address Generation

7.3.1.2.1 Reading

Address Generation for reading is straightforward. Each cycle we generate a bit address which is used to fetch 4 bits representing 1-bit per color for the particular segment. By adding 400 to the current bit address, we advance to the next segment's equivalent dot. We add 400 (not 800) since the odd and even dots are separated in the buffer. We do this 16 times to retrieve the two sets of 32 bits for the two sets of 8 segments representing the even dots (the resultant data is transferred to the MJI 32 bits at a time) and another 16 times to load the odd dots. This 32-cycle process is repeated 400 times, incrementing the start address each time. Thus in 400 × 32 cycles, a total of 400 × 32 × 4 (51,200) dot values are transferred in the order required by the printhead.

In addition, we generate the TransferWriteEnable control signal. Since the LLFU starts before the MJI, we must transfer the first value before the Advance pulse from the MJI. We must also generate the next 32-bit value in readiness for the first Advance pulse. The solution is to transfer the first 32-bit value to the Transfer register after 8 cycles, and then to stall 8-cycles later, waiting for the Advance pulse to start the next 8-cycle group. Once the first Advance pulse arrives, the LLFU is synchronized to the MJI. However, the MJI

must be started at least 16 cycles after the LLFU so that the initial Transfer value is valid and the next 32-bit value is ready to be loaded into the Transfer register.

The read process is shown in the following pseudocode:

```
DotCount = 0
For DotInSegment0 = 0 to 400
   CurrAdr = DotInSegment0
   Do
      V1 = (CurrAdr=0) OR (CurrAdr=3200)
      V2 = Low 3 bits of DotCount = 0
      TransferWriteEnable = V1 OR ADVANCE
      Stall = V2 AND (NOT TransferWriteEnable)
      If (NOT Stall)
         Shift Register=Fetch 4-bits from CurrReadBuffer:CurrAdr
         CurrAdr = CurrAdr + 400
         DotCount = (DotCount + 1) MOD 32(odd&even, printheads 1&2, segments 0-7)
      EndIf
   Until (DotCount=0) AND (NOT Stall)
EndFor
```

Once the line has finished, the CurrReadBuffer value must be toggled.

7.3.1.2.2 Writing

The write process is also straightforward. 4 DMA request lines are output to the DMA Controller. As requests are satisfied by the return DMA Acknowledge lines, the appropriate 8-bit destination address is selected (the lower 5 bits of the 13-bit output address are *don't care* values) and the acknowledge signal is passed to the correct buffer's WriteEnable control line (the Current Write Buffer is ¬CurrentReadBuffer). The 8-bit destination address is selected from the 4 current addresses, one address per color. As DMA requests are satisfied the appropriate destination address is incremented, and the corresponding TransfersRemaining counter is decremented. The DMA request line is only set when the number of transfers remaining for that color is non-zero.

The following pseudocode illustrates the Write process:

```
CurrentAdr[0-3] = 0
While (TransfersRemaining[0-3] are all non-zero)
   DMARequest[0-3] = TransfersRemaining[0-3] != 0
   If DMAAknowledge[N]
      CurrWriteBuffer:CurrentAdr[N] = Fetch 32-bits from data bus
      CurrentAdr[N] = CurrentAdr[N] + 1
      TransfersRemaining[N] = TransfersRemaining[N] - 1 (floor 0)
   EndIf
EndWhile
```

7.3.1.3 Registers

The following registers are contained in the LLFU:

Table 25. Line Load/Format Unit Registers

Register Name	Description
CurrentReadBuffer	The current buffer being read from. When Buffer0 is being read from, Buffer1 is written to and vice versa. Should be toggled with each AdvanceLine pulse from the MJI.
Go	Bits 0 and 1 control the starting of the read and write processes respectively. A non-zero write to the appropriate bit starts the process.
Stop	Bits 0 and 1 control the stopping of the read and write processes respectively. A non-zero write to the appropriate bit stops the process.
TransfersRemainingC	The number of 32-bit transfers remaining to be read into the Cyan buffer
TransfersRemainingM	The number of 32-bit transfers remaining to be read into the Magenta buffer
TransfersRemainingY	The number of 32-bit transfers remaining to be read into the Yellow buffer
TransfersRemainingK	The number of 32-bit transfers remaining to be read into the Black buffer

7.3.2 Memjet Interface

The Memjet Interface (MJI) transfers data to the Memjet printhead, and controls the nozzle firing sequences during a print.

The MJI is simply a State Machine (see Figure 37) which follows the Printhead loading and firing order described in Section 6.1.2 on page 30, and includes the functionality of the Preheat Cycle and Cleaning Cycle as described in Section 6.1.4 on page 33 and Section 6.1.5 on page 34. Both high-speed and low-speed printing modes are available.

The MJI loads data into the printhead from a choice of 2 data sources:

- All 1s. This means that all nozzles will fire during a subsequent Print cycle, and is the standard mechanism for loading the printhead for a preheat or cleaning cycle.
- From the 32-bit input held in the Transfer register of the LLFU. This is the standard means of printing an image. The 32-bit value from the LLFU is directly sent to the printhead and a 1-bit 'Advance' control pulse is sent to the LLFU. At the end of each line, a 1-bit 'AdvanceLine' pulse is also available.

The MJI must be started after the LLFU has already prepared the first 32-bit transfer value. This is so the 32-bit data input will be valid for the first transfer to the printhead.

The MJI is therefore directly connected to the LLFU and the external Memjet printhead. The basic structure is shown in Figure 37.

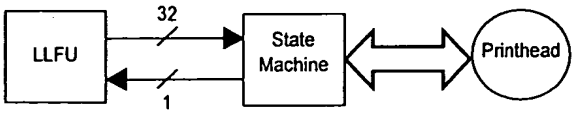


Figure 37. Memjet Interface

7.3.2.1 *Connections to Printhead*

The MJI has the following connections to the printhead, with the sense of input and output with respect to the MJI. The names match the pin connections on the printhead (see Section 6.2.1 on page 36 for an explanation of the way the 8-inch printhead is wired up).

Table 26. Memjet Interface Connections

Name	#Pins	I/O	Description
Quadpod Select	3	O	Select which quadpod will fire (0-4)
NozzleSelect	4	O	Select which nozzle from the pod will fire (0-9)
PodgroupEnable	2	O	Enable the podgroups to fire (choice of: 01, 10, 11)
AEnable	1	O	Firing pulse for podgroup A
BEnable	1	O	Firing pulse for podgroup B
CDataIn[0-7]	8	O	Cyan output to cyan shift register of segments 0-7
MDataIn[0-7]	8	O	Magenta input to magenta shift register of segments 0-7
YDataIn[0-7]	8	O	Yellow input to yellow shift register of segments 0-7
KDataIn[0-7]	8	O	Black input to black shift register of segment 0-7
SRClock1	1	O	A pulse on SRClock1 (ShiftRegisterClock1) loads the current values from CDataIn[0-7], MDataIn[0-7], YDataIn[0-7] and KDataIn[0-7] into the 32 shift registers of 4-inch printhead 1
SRClock2	1	O	A pulse on SRClock2 (ShiftRegisterClock2) loads the current values from CDataIn[0-7], MDataIn[0-7], YDataIn[0-7] and KDataIn[0-7] into the 32 shift registers of 4-inch printhead 2
PTransfer	1	O	Parallel transfer of data from the shift registers to the printhead's internal NozzleEnable bits (one per nozzle).
SenseEnable1	1	O	A pulse on SenseEnable1 ANDed with data on CDataIn[n] enables the sense lines for segment n in 4-inch printhead 1.
SenseEnable2	1	O	A pulse on SenseEnable2 ANDed with data on CDataIn[n] enables the sense lines for segment n in 4-inch printhead 2.
Tsense	1	I	Temperature sense
Vsense	1	I	Voltage sense
Rsense	1	I	Resistivity sense
Wsense	1	I	Width sense
TOTAL	52		

7.3.2.2 *Registers*

The Processor communicates with the MJI via a register set. The registers allow the Processor to parameterize a print as well as receive feedback about print progress.

The following registers are contained in the MJI:

Table 27. Memjet Interface Registers

Register Name	Description
Print Parameters	
NumTransfers	The number of transfers required to load the Printhead (usually 1600). This is the number of pulses for *both* SRClock lines and the total number of 32-bit data values to transfer for a given line.
PulseDuration	Fixed point number to determine the duration of a single pulse on the ColorEnable lines. Duration range = 0-6 µs. Typical range = 1.3-1.8 µs.
PrintSpeed	Whether to print at low or high speed (determines the value on the PodgroupEnable lines during the print).
NumLines	The number of Load/Print cycles to perform.
Monitoring the Print	
Status	The Memjet Interface's Status Register
LinesRemaining	The number of lines remaining to be printed. Only valid while Go=1. Starting value is NumLines.
TransfersRemaining	The number of transfers remaining before the Printhead is considered loaded for the current line. Only valid while Go=1.
SenseSegment	The 8-bit value to place on the Cyan data lines during a subsequent feedback SenseEnable pulse. Only 1 of the 8 bits should be set, corresponding to one of the 8 segments. See SenseEnable for how to determine which of the two 4-inch printheads to sense.
SetAllNozzles	If non-zero, the 32-bit value written to the printhead during the LoadDots process is all 1s, so that all nozzles will be fired during the subsequent PrintDots process. This is used during the preheat and cleaning cycles. If 0, the 32-bit value written to the printhead comes from the LLFU. This is the case during the actual printing of regular images.
Actions	
Reset	A write to this register resets the MJI, stops any loading or printing processes, and loads all registers with 0.
SenseEnable	A write to this register with any value clears the Feedback bit of the Status register, and depending on the low-order bit, sends a pulse on the SenseEnable1 or SenseEnable2 line if the LoadingDots and PrintingDots status bits are all 0. If any of the status bits are set, the Feedback bit is cleared and nothing more is done. Once the various sense lines have been tested, the values are placed in the Tsense, Vsense, Rsense, and Wsense registers, and then the Feedback bit of the Status register is set.
Go	A write of 1 to this bit starts the LoadDots / PrintDots cycles. A total of NumLines lines are printed, each containing NumTransfers 32 bit transfers. As each line is printed, LinesRemaining decrements, and TransfersRemaining is reloaded with NumTransfers again. The status register contains print status information. Upon completion of NumLines, the loading/printing process stops and the Go bit is cleared. During the final print cycle, nothing is loaded into the printhead. A write of 0 to this bit stops the print process, but does not clear any other registers.

Table 27. Memjet Interface Registers

Register Name	Description
Tsense	Read only feedback of Tsense from the last SenseEnable pulse sent to segment SenseSegment. Is only valid if the Feedback bit of the Status register is set.
Vsense	Read only feedback of Vsense from the last SenseEnable pulse sent to segment SenseSegment. Is only valid if the Feedback bit of the Status register is set.
Rsense	Read only feedback of Rsense from the last SenseEnable pulse sent to segment SenseSegment. Is only valid if the Feedback bit of the Status register is set.
Wsense	Read only feedback of Wsense from the last SenseEnable pulse sent to segment SenseSegment. Is only valid if the Feedback bit of the Status register is set.

The MJI's Status Register is a 16-bit register with bit interpretations as follows:

Table 28. MJI Status Register

Name	Bits	Description
LoadingDots	1	If set, the MJI is currently loading dots, with the number of dots remaining to be transferred in TransfersRemaining. If clear, the MJI is not currently loading dots
PrintingDots	1	If set, the MJI is currently printing dots. If clear, the MJI is not currently printing dots.
PrintingA	1	This bit is set while there is a pulse on the AEnable line
PrintingB	1	This bit is set while there is a pulse on the BEnable line
PrintingQuadpod	4	This holds the current quadpod being fired while the PrintingDots status bit is set.
PrintingNozzles	4	This holds the current nozzle being fired while the PrintingDots status bit is set.
Reserved	4	-

7.3.2.3 *Preheat and Cleaning Cycles*

The Cleaning and Preheat cycles are simply accomplished by setting appropriate registers:

- SetAllNozzles = 1
- Set the PulseDuration register to either a low duration (in the case of the preheat mode) or to an appropriate drop ejection duration for cleaning mode.
- Set NumLines to be the number of times the nozzles should be fired
- Set the Go bit and then wait for the Go bit to be cleared when the print cycles have completed.

7.4 PROCESSOR AND MEMORY

7.4.1 Processor

The Processor runs the control program which synchronises the other functional units during page reception, expansion and printing. It also runs the device drivers for the various external interfaces, and responds to user actions through the user interface.

It must have low interrupt latency, to provide efficient DMA management, but otherwise does not need to be particularly high-performance.

7.4.2 DMA Controller

The DMA Controller supports single-address transfers on 26 channels (see Table 29). It generates vectored interrupts to the Processor on transfer completion.

Table 29. DMA channel usage

functional unit	input channels	output channels
USB Interface	-	1
EDRL Bi-Level Page Expander	1	1
JPEG Codec	1	8
Halftoner/Compositor	2	8
Printhead Interface	4	-
	8	18
		26

7.4.3 Program ROM

The program ROM holds the ICP control program which is loaded into main memory during system boot.

7.4.4 Rambus Interface

The Rambus interface provides the high-speed interface to the external 8MB (64Mbit) Rambus DRAM (RDRAM).

7.5 EXTERNAL INTERFACES

7.5.1 USB Interface

The Universal Serial Bus (USB) Interface provides a standard USB device interface.

7.5.2 Parallel Interface

The Parallel Interface provides i/o on a number of parallel external signal lines.

It allows the Processor to sense or control the devices listed in Table 30.

Table 30. Parallel Interface devices

parallel interface devices
power button
paper feed button
power LED
paper jam LED
ink low LED
media sensor
capping solenoid

Table 30. Parallel Interface devices

parallel interface devices
paper transport stepper motor
speaker

7.5.3 Serial Interface

The Serial Interface provides two standard low-speed serial ports.

One port is used to connect to the system Authentication Chip. The other is used to connect to the Authentication Chip in the ink cartridge. The Processor-mediated protocol between the two is used to authenticate the ink cartridge [15,16].

7.5.4 JTAG Interface

A standard JTAG (Joint Test Action Group) Interface is included for testing purposes. Due to the complexity of the chip, a variety of testing techniques are required, including BIST (Built In Self Test) and functional block isolation. An overhead of 10% in chip area is assumed for overall chip testing circuitry. The test circuitry is beyond the scope of this document.

8 Generic Printer Driver

This section describes generic aspects of any host-based printer driver for iPrint.

8.1 Graphics and Imaging Model

We assume that the printer driver is closely coupled with the host graphics system, so that the printer driver can provide device-specific handling for different graphics and imaging operations, in particular compositing operations and text operations.

We assume that the host provides support for color management, so that device-independent color can be converted to iPrint-specific CMYK color in a standard way, based on a user-selected iPrint-specific ICC (International Color Consortium) color profile. The color profile is normally selected implicitly by the user when the user specifies the output medium in the printer (i.e. plain paper, coated paper, transparency, etc.). The page description sent to the printer always contains *device-specific* CMYK color.

We assume that the host graphics system renders images and graphics to a nominal resolution specified by the printer driver, but that it allows the printer driver to take control of rendering text. In particular, that the graphics system provides sufficient information to the printer driver to allow it to render *and position* text at a higher resolution than the nominal device resolution.

We assume that the host graphics system requires random access to a contone page buffer at the nominal device resolution, into which it composites graphics and imaging objects, but that it allows the printer driver to take control of the actual compositing - i.e. it allows the printer driver to *manage* the page buffer.

8.2 Two-Layer Page Buffer

The printer has a contone resolution of 267 ppi and a black resolution of 800 dpi. The printer driver therefore specifies a nominal page resolution of 267 ppi to the graphics system. Where possible the printer driver relies on the graphics system to render image and graphics objects to the pixel level at 267 ppi, with the exception of *black* text. The printer driver fields all text rendering requests, detects and renders black text at 800 dpi, but returns non-black text rendering requests to the graphics system for rendering at 267 ppi.

Ideally the graphics system and the printer driver manipulate color in device-independent RGB, deferring conversion to device-specific CMYK until the page is complete and ready to be sent to the printer. This reduces page buffer requirements and makes compositing more rational. Compositing in CMYK color space is not ideal.

Ultimately the graphics system asks the printer driver to composite each rendered object into the printer driver's page buffer. Each such object uses 24-bit contone RGB, and has an explicit (or implicitly opaque) opacity channel.

The printer driver maintains a three-part page buffer. The first part is the low-resolution (267 ppi) contone layer. This consists of a 24-bit RGB bitmap. The second part is a low-resolution black layer. This consists of an 8-bit opacity bitmap. The third part is a high-resolution (800 dpi) black layer. This consists of a 1-bit opacity bitmap. The low-resolution black layer is a subsampled version of the high-resolution opacity layer. In practice, assuming the low resolution is an integer factor n of the high resolution (e.g. $n = 800 / 267 = 3$), each low-resolution opacity value is obtained by averaging the corresponding $n \times n$

high-resolution opacity values. This corresponds to box-filtered subsampling. The subsampling of the black pixels effectively antialiases any edges, thereby reducing ringing artifacts when the contone layer is subsequently JPEG compressed and decompressed.

The structure and size of the page buffer is illustrated in Figure 38.

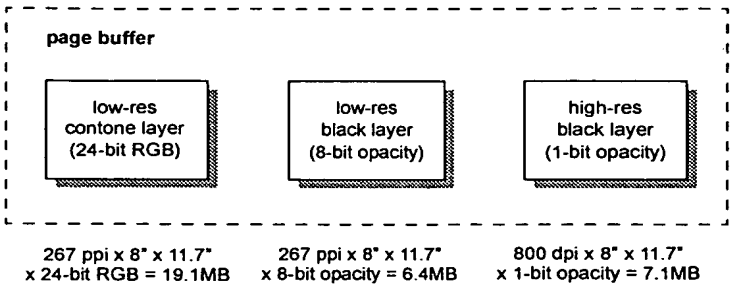


Figure 38. Two-layer page buffer

8.3 COMPOSITING MODEL

For the purposes of discussing the page buffer compositing model, we define the following variables.

Table 31. Compositing variables

variable	description	resolution	format
n	low to high resolution scale factor	-	-
C_C	contone layer color	low	8-bit color component
C_G	contone object color	low	8-bit color component
α_G	contone object opacity	low	8-bit opacity
α_L	low-resolution black layer opacity	low	8-bit opacity
α_B	black layer opacity	high	1-bit opacity
α_T	black object opacity	high	1-bit opacity

When a black object of opacity α_T is composited with the black layer, the black layer is updated as follows:

$$\alpha_B = \alpha_B \vee \alpha_T \quad (1)$$

$$\alpha_L = \frac{1}{n^2}\sum_{i}^{n}\sum_{j}^{n}(\alpha_B \times 255) \quad (2)$$

The object opacity is simply *or*ed with the black layer opacity (Eq. 1), and the corresponding part of the low-resolution black layer is re-computed from the high-resolution black layer (Eq. 2).

When a contone object of color C_G and opacity α_G is composited with the contone layer, the contone layer and the black layer are updated as follows:

$$C_C = C_C(1-\alpha_L) \text{ if } (\alpha_G > 0) \quad (3)$$

$$\alpha_L = 0 \text{ if } (\alpha_G > 0) \quad (4)$$

$$\alpha_B = 0 \text{ if } (\alpha_G > 0) \quad (5)$$

$$C_C = C_C(1-\alpha_G) + C_G\alpha_G \quad (6)$$

Wherever the contone object hides the black layer, even if not fully opaquely, the affected black layer pixels are composited with the contone layer (Eq. 3) and removed from the black layer (Eq. 4 and Eq. 5). The contone object is then composited with the contone layer (Eq. 6).

8.4 PAGE COMPRESSION AND DELIVERY

Once page rendering is complete, the printer driver converts the contone layer to iPrint-specific CMYK with the help of color management functions in the graphics system.

The printer driver then compresses and packages the black layer and the contone layer into an iPrint page description as described in Section 5.2. This page description is delivered to the printer via the standard spooler.

9 Windows 9x/NT Printer Driver

9.1 Windows 9x/NT Printing System

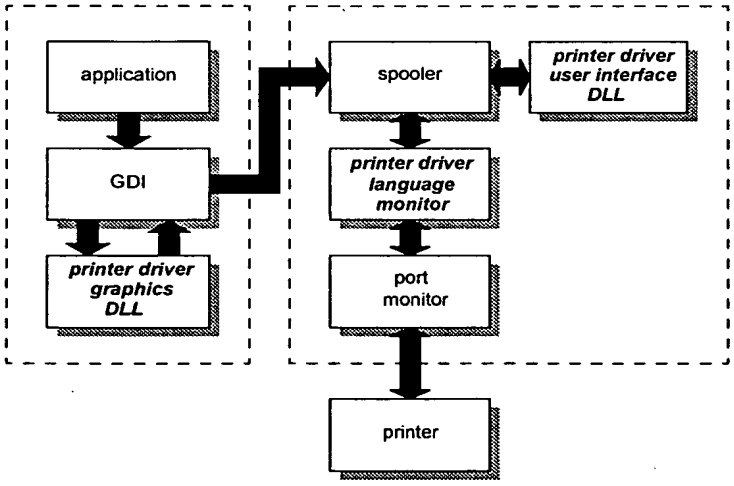
In the Windows 9x/NT printing system [9][10], a printer is a *graphics device*, and an application communicates with it via the *graphics device interface* (GDI). The printer driver *graphics DLL* (dynamic link library) implements the device-dependent aspects of the various graphics functions provided by GDI.

The *spooler* handles the delivery of pages to the printer, and may reside on a different machine to the application requesting printing. It delivers pages to the printer via a *port monitor* which handles the physical connection to the printer. The optional *language monitor* is the part of the printer driver which imposes additional protocol on communication with the printer, and in particular decodes status responses from the printer on behalf of the spooler.

The printer driver *user interface DLL* implements the user interface for editing printer-specific properties and reporting printer-specific events.

The structure of the Windows 9x/NT printing system is illustrated in Figure 39.

Figure 39. Windows 9x/NT printing system with printer driver components indicated



Since iPrint uses USB IEEE-1284 emulation, there is no need to implement a language monitor for iPrint.

The remainder of this section describes the design of the printer driver graphics DLL. It should be read in conjunction with the appropriate Windows 9x/NT DDK documentation [9][10].

9.2 Windows 9x/NT Graphics Device Interface (GDI)

GDI provides functions which allow an application to draw on a *device surface*, i.e. typically an abstraction of a display screen or a printed page. For a raster device, the device surface is conceptually a color bitmap. The application can draw on the surface in a

device-independent way, i.e. independently of the resolution and color characteristics of the device.

The application has random access to the entire device surface. This means that if a memory-limited printer device requires banded output, then GDI must buffer the entire page's GDI commands and replay them windowed into each band in turn. Although this provides the application with great flexibility, it can adversely affect performance.

GDI supports color management, whereby device-independent colors provided by the application are transparently translated into device-dependent colors according to a standard ICC (International Color Consortium) color profile of the device. A printer driver can activate a different color profile depending, for example, on the user's selection of paper type on the driver-managed printer property sheet.

GDI supports line and spline outline graphics (paths), images, and text. Outline graphics, including outline font glyphs, can be stroked and filled with bit-mapped brush patterns. Graphics and images can be geometrically transformed and composited with the contents of the device surface. While Windows 95/NT4 provides only boolean compositing operators, Windows 98/NT5 provides proper alpha-blending [10].

9.3 PRINTER DRIVER GRAPHICS DLL

A raster printer can, in theory, utilize standard printer driver components under Windows 9x/NT, and this can make the job of developing a printer driver trivial. This relies on being able to model the device surface as a single bitmap. The problem with this is that text and images must be rendered at the same resolution. This either compromises text resolution, or generates too much output data, compromising performance.

As described earlier, iPrint's approach is to render black text and images at different resolutions, to optimize the reproduction of each. The printer driver is therefore implemented according to the generic design described in Section 8.

The driver therefore maintains a two-layer three-part page buffer as described in Section 8.2, and this means that the printer driver must take over managing the device surface, which in turn means that it must mediate all GDI access to the device surface.

9.3.1 Managing the Device Surface

The printer driver must support a number of standard functions, including the following:

Table 32. Standard graphics driver interface funtions

function	description
`DrvEnableDriver`	Initial entry point into the driver graphics DLL. Returns addresses of functions supported by the driver.
`DrvEnablePDEV`	Creates a logical representation of a physical device with which the driver can associate a drawing surface.
`DrvEnableSurface`	Creates a surface to be drawn on, associated with a given PDEV.

`DrvEnablePDEV` indicates to GDI, via the `flGraphicsCaps` member of the returned `DEVINFO` structure, the graphics rendering capabilities of the driver. This is discussed further below.

`DrvEnableSurface` creates a device surface consisting of two conceptual layers and three parts: the 267 ppi contone layer 24-bit RGB color, the 267 ppi black layer 8-bit opacity, and the 800 dpi black layer 1-bit opacity. The *virtual* device surface which encapsulates these two layers has a nominal resolution of 267 ppi, so this is the resolution at which GDI operations take place.

Although the aggregate page buffer requires about 33MB of memory, the PC 99 office standard [6] specifies a minimum of 64MB.

In practice, managing the device surface and mediating GDI access to it means that the printer driver must support the following additional functions:

Table 33. Required graphics driver functions for a device-managed surface

function	description
`DrvCopyBits`	Translates between device-managed raster surfaces and GDI-managed standard-format bitmaps.
`DrvStrokePath`	Srokes a path.
`DrvPaint`	Paints a specified region.
`DrvTextOut`	Renders a set of glyphs at specified positions.

Copying images, stroking paths and filling regions all occur on the contone layer, while rendering solid black text occurs on the bi-level black layer. Furthermore, rendering non-black text also occurs on the contone layer, since it isn't supported on the black layer. Conversely, stroking or filling with solid black can occur on the black layer (if we so choose).

Although the printer driver is obliged to *hook* the aforementioned functions, it can *punt* function calls which apply to the contone layer back to the corresponding GDI implementations of the functions, since the contone layer is a standard-format bitmap. For every `DrvXxx` function there is a corresponding `EngXxx` function provided by GDI.

Graphics objects are laid down on the device surface bottom-up - i.e. later objects obscure earlier objects. This works naturally when there is only a single surface, but not when there are two surfaces which will be composited later. It is therefore necessary to detect when an object being placed on the contone layer obscures something on the black layer.

When obscuration is detected, the obscured black pixels are composited with the contone layer and removed from the black layer. The obscuring object if then laid down on the contone layer, possibly interacting with the black pixels in some way. If the compositing mode of the obscuring object is such that no interaction with the background is possible, then the black pixels can simply be discarded without being composited with the contone layer.

The key to this process working is that obscuration is detected and handled in the hooked call, *before* it is punted back to GDI.

9.3.2 Detecting Black Layer Obscuration

It is possible to determine the geometry of each contone object before it is rendered and thus determine efficiently which black pixels it obscures. In the case of `DrvCopyBits` and `DrvPaint`, the geometry is determined by a clip object (`CLIPOBJ`), which can be enumerated as a set of rectangles.

In the case of DrvStrokePath, things are more complicated. DrvStrokePath supports both straight-line and Bézier-spline curve segments, and single-pixel-wide lines and geometric-wide lines. The first step is to avoid the complexity of Bézier-spline curve segments and geometric-wide lines altogether by clearing the corresponding capability flags (GCAPS_BEZIERS and GCAPS_GEOMETRICWIDE) in the flGraphicsCaps member of the driver's DEVINFO structure. This causes GDI to reformulate such calls as sets of simpler calls to DrvPaint. In general, GDI gives a driver the opportunity to accelerate high-level capabilities, but *simulates* any capabilities not provided by the driver.

What remains is simply to determine the geometry of a single-pixel-wide staight line. Such a line can be solid or *cosmetic*. In the latter case, the line style is determined by a styling array in the specified line attributes (LINEATTRS). The styling array specifies how the line alternates between being opaque and transparent along its length, and so supports various dashed line effects etc.

When the brush is solid black, straight lines can also usefully be rendered to the black layer, though with the increased width implied by the 800 dpi resolution.

9.3.3 Rendering Text

In the case of a DrvTextOut, things are also more complicated. Firstly, the opaque background, if any, is handled like any other fill on the contone layer (see DrvPaint). If the foreground brush is not black, or the mix mode is not effectively opaque, or the font is not scalable, or the font indicates outline stroking, then the call is punted to EngTextOut, to be applied to the contone layer. Before the call is punted, however, the driver determines the geometry of each glyph by obtaining its bitmap (via FONTOBJ_cGetGlyphs), and makes the usual obscuration check against the black layer.

If punting a DrvTextOut call is not allowed (the documentation is ambiguous), then the driver should disallow complex text operations. This includes disallowing outline stroking (by clearing the GCAPS_VECTOR_FONT capability flag), and disallowing complex mix modes (by clearing the GCAPS_ARBMIXTXT capability flag).

If the foreground brush is black and opaque, and the font is scalable and not stroked, then the glyphs are rendered on the black layer. In this case the driver determines the geometry of each glyph is by obtaining its *outline* (again via FONTOBJ_cGetGlyphs, but as a PATHOBJ). The driver then renders each glyph from its outline at 800 dpi and writes it to the black layer. Although the outline geometry uses device coordinates (i.e. at 267 ppi), the coordinates are in fixed point format with plenty of fractional precision for higher-resolution rendering.

Note that strikethrough and underline rectangles are added to the glyph geometry, if specified.

The driver must set the GCAPS_HIGHRESTEXT flag in the DEVINFO to request that glyph positions (again in 267 ppi device coordinates) be supplied by GDI in high-precision fixed-point format, to allow accurate positioning at 800 dpi. The driver must also provide an implementation of the DrvGetGlyphMode function, so that it can indicate to GDI that glyphs should be cached as outlines rather than bitmaps. Ideally the driver should cache rendered glyph bitmaps for efficiency, memory allowing. Only glyphs below a certain point size should be cached.

9.3.4 Compressing the Contone Layer

As described earlier, the contone layer is compressed using JPEG. The forward discrete cosine transform (DCT) is the costliest part of JPEG compression. In current high-quality software implementations, the forward DCT of each 8×8 block requires 12 integer multiplications and 32 integer additions [8]. On a Pentium processor, an integer multiplication requires 10 cycles, and an integer addition requires 2 cycles [14]. This equates to a total cost per block of 184 cycles.

The 25.5MB contone layer consists of 417,588 JPEG blocks, giving an overall forward DCT cost of about 77Mcycles. At 300MHz, the PC 99 desktop standard [6], this equates to 0.26 seconds, which is well within the 2 second limit per page.

10 References

[1] ANSI/EIA 538-1988, *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Equipment*, August 1988

[2] Humphreys, G.W., and V. Bruce, *Visual Cognition*, Lawrence Erlbaum Associates, 1989, p.15

[3] IEEE Std 1284-1994, *IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers*, 2 December 1994

[4] IEEE Std 1284.1-1997, IEE Standard for Information Technology - Transport Independent Printer/System Interface (TIP/SI), 20 October 1997

[5] Intel Corp. and Microsoft Corp., *PC 98 System Design Guide*, 1997

[6] Intel Corp. and Microsoft Corp., *PC 99 System Design Guide*, 1998

[7] ISO/IEC 19018-1:1994, *Information technology - Digital compression and coding of continuous-tone still images: Requirements and guidelines*, 1994

[8] Loeffler, C., A. Ligtenberg and G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications", Proceedings of the International Conference on Acoustics, Speech, and Signal Processing 1989 (ICASSP '89), pp.988-991

[9] Microsoft Corp., *Microsoft Windows NT 4.0 Device Driver Kit*, 1997

[10] Microsoft Corp., *Microsoft Windows NT 5.0 Device Driver Kit*, 1998

[11] Murray, J.D., and van Ryper, W., *Encyclopedia of Graphics File Formats*, O'Reilly & Associates, First Ed., 1994

[12] Olsen, J. "Smoothing Enlarged Monochrome Images", in Glassner, A.S. (ed.), *Graphics Gems*, AP Professional, 1990

[13] Pennebaker, W.B., and Mitchell, J.L., *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, September 1992

[14] Schmit, M.L., *Pentium Processor Optimization Tools*, AP Professional, 1995

[15] Silverbrook Research, *Authentication Chip*, 1998

[16] Silverbrook Research, *Authentication of Consumables*, 1998

[17] Silverbrook Research, *Memjet*, 1998

[18] Thompson, H.S., *Multilingual Corpus 1* CD-ROM, European Corpus Initiative

[19] Urban, S.J., "Review of standards for electronic imaging for facsimile systems", Journal of Electronic Imaging, Vol.1(1), January 1992, pp.5-21

[20] USB Implementers Forum, *Universal Serial Bus Specification*, Revision 1.0, 1996

[21] USB Implementers Forum, *Universal Serial Bus Device Class Definition for Printer Devices*, Version 1.07 Draft, 1998

[22] Wallace, G.K., "The JPEG Still Picture Compression Standard", *Communications of the ACM*, 34(4), April 1991, pp.30-44

[23] Yasuda, Y., "Overview of Digital Facsimile Coding Techniques in Japan", Proceedings of the IEEE, Vol. 68(7), July 1980, pp.830-845